A Comparative Study of Query Execution Times: MongoDB vs PostgreSQL

Samiksha Burkul , Rachel Culbreath, Krishnasurya Gopalakrishnan

Data Science, George Washington University
2121 I St NW, Washington, DC 20052

samiksharamnath.burkul@gwu.edu

rachel.culbreath@gwu.edu

krishnasurya.gopalakrishnan@gwu.edu

***Abstract*— This paper describes the test and application of queries within MongoDB and Postgre. This comparative analysis of the time it takes to complete a query allows us to uncover differences within the performance of the database systems. To achieve a robust understanding of the topic we utilized entry, intermediate, and advance queries to illustrate the methodology that sets the systems apart.**

I. Introduction

For this experiment we selected two database systems in order to test the efficiency of non-relational and relational databases. These tests will allow us to effectively make queries and gain an insight into the efficiency of database systems. This study aims to explore the execution time of PostgreSQL and MongoDB. Through this we can reflect on the strengths and weaknesses in handling basic, intermediate, and advanced queries. Utilizing basic queries will allow us to gain insight into how it handles basic commands and by starting at a basic query model we are able to add additional requirements to infer how it affects the time. Intermediate queries will allow us to ask questions that test the optimization time of MongoDB and PostgreSQL. The question allows us to perform tests that showcase the understanding of the application of the two databases. The way the database stores information differs and we will test to see if this contributes to the time it takes to execute the query. The advanced query will test to see how a database handles queries with multiple requirements and performing all of these queries will allow us to gain a robust understanding of the time execution differences.

II. MongoDB

MongoDB is an open-source document database built on a horizontal scale-out architecture that uses a flexible schema for storing data. Founded in 2007, MongoDB has a worldwide following in the developer community. MongoDB is a leading NoSQL database management system, renowned for its flexibility and scalability in handling unstructured or semi-structured data. It adopts a document-oriented data model, storing data in flexible JSON-like documents within collections, allowing for dynamic schema changes and ease of data manipulation. MongoDB leverages high performance and horizontal scalability, making it ideal for applications requiring agile development, real-time analytics, and rapid iterations.

III. PostgreSQL

PostgreSQL is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads. The origins of PostgreSQL date back to 1986 as part of the POSTGRES project at the University of California at Berkeley and has more than 35 years of active development on the core platform. It employs a traditional table-based structure with support for complex queries, transactions, and ACID (Atomicity, Consistency, Isolation, Durability)

compliance, ensuring data integrity. PostgreSQL offers a rich set of features, including extensibility through user-defined functions, numerous indexing techniques, and a reputation for reliability, making it a preferred choice for data-intensive applications, complex queries, and enterprise-level deployments.

III. METHODOLOGY

The methodology for this project consisted of exploration of the topic of MongoDB and PostgreSQL. We conducted research in order to understand the best practices for comparing the data accurately. We explored the data set and selected variables that would allow us to best test the speed and accuracy of the different types of databases. In order to ensure that the tests were performed correctly we took the same data and performed the same test in MongoDB and PostgreSQL. We drafted and tested our queries in PostgreSQL as it was easy to understand based on the user readable format. Our group had a basic understanding of SQL so drafting and executing basic, intermediate, and advanced queries in PostgreSQL allowed us to determine if they would be good queries to test between the two databases. The selection process took place and once that was completed we began to translate our selected queries into MongoDB. SQLAlchemy is used to connect to PostgreSQL DB via python and PyMongo is used to connect to MongoDB via Python. Queries are run via python scripts in both. The execution time is recorded by taking the difference of the timings recorded at the start and end of each query by using the time library. The queries are arranged in increasing order of difficulty and their respective time taken in PostgreSQL and MongoDB are plotted as two line plots to observe the time taken between the two databases effectively.

I) Queries Execution :

Query 1: Total cities that are Waterloo:

MongoDB:

```
count = ls.count_documents({"city": "Waterloo"})
```

PostgreSQL:

```
SELECT count(*) FROM ls WHERE city = 'Waterloo';
```

In MongoDB, count_document() method used along with the condition to match the “city” to “Waterloo” is used.

In PostgreSQL, count(*) which means count all performed along with the WHERE condition matching the “city” to “Waterloo” is used.

Query 2: Retrieving invoices within a specified sale amount range:

MongoDB:

```
results = ls.find({"sale_usd": {"$gte": 500, "$lte": 5000}}, {"invoice_number": 1, "date": 1, "store_name": 1, "sale_usd": 1})
```

PostgreSQL:

```
SELECT invoice_number, date, store_name, sale_usd
FROM ls
WHERE sale_usd
BETWEEN 500 AND 5000;
```

In MongoDB, the find() method is used to retrieve documents that match the specified condition which is gte (greater than) and lte (less than) the set amounts. In the second part of the find() method, we can specify which fields need to be displayed in the result by assigning 1 to those fields (0 can only be assigned to the “_id” field)

Query 3: Profitable years and the corresponding bottles sold:

MongoDB:

```
pipeline = [
   {
      "$group": {
         "_id": {"$year": "$date"},
         "total_bottles_sold": {"$sum": "$bottles_sold"},
         "total_sale_usd": {"$sum": "$sale_usd"}
      }
```

```
    },
    {
        "$sort": {"total_sale_usd": -1}
    }
]
result = ls.aggregate(pipeline)
```

PostgreSQL:

```
SELECT EXTRACT(YEAR FROM date)
AS year,
SUM(bottles_sold) AS total_bottles_sold,
SUM(sale_usd) AS total_sale_usd
FROM ls
GROUP BY year
ORDER BY SUM(sale_usd) DESC;
```

In MongoDB, the above aggregation pipeline groups sales data by year from the date attribute and group is used to aggregate and sum the sales_usd field values over each year and sort is applied on the aggregated field with a "-1" parameter for descending order.

In PostgreSQL, EXTRACT(YEAR FROM date_column) is used to extract the year information from a date column. After year is obtained for each record, the total sales aggregation is obtained by using GROUP BY year obtained and ORDER BY is used on the aggregated column by specifying the aggregation formula again and DESC is used to arrange the results in the descending order.

Query 4 Total Packs Sold per Store:

MongoDB:

```
pipeline = [
    {"$group": {"_id": "$store_number",
"pack": {"$sum": "$pack"}}},
    {"$sort": {"pack": -1}}
]
results = ls.aggregate(pipeline)
```

PostgreSQL:

```
SELECT store_number, SUM(pack) AS
pack
FROM ls
GROUP BY store_number
ORDER BY SUM(pack) DESC;
```

In MongoDB, aggregation is applied as mentioned in the previous query explanation.

In PostgreSQL, aggregation is again applied as mentioned earlier in the previous query.

Query 5: Vendors with significant sales:

MongoDB:

```
pipeline = [
    {
        "$group": {
            "_id": {
                "store_number": "$store_number",
                "vendor_number":
"$vendor_number"
            },
            "bottles_sold": {"$sum":
"$bottles_sold"},
            "sales_usd": {"$sum": "$sale_usd"}
        }
    },
    {
        "$match": {
            "bottles_sold": {"$gt": 1000},
            "sales_usd": {"$gt": 500000}
        }
    },
    {
        "$sort": {"_id.store_number": 1,
"_id.vendor_number": -1}
    }
]
result = ls.aggregate(pipeline)
```

PostgreSQL:

```
SELECT store_number,
vendor_number, SUM(bottles_sold) AS
total_bottles_sold, SUM(sale_usd) AS
total_sales_usd
FROM ls
GROUP BY store_number, vendor_number
```

```
HAVING SUM(bottles_sold) > 1000
AND SUM(sale_usd) > 500000
ORDER BY store_number, vendor_number
DESC;
```

The idea behind identifying the specific vendors is to filter out the vendors that have sales over 50,000 USD and bottles sold over 1000.

In MongoDB, like before, group is used to aggregate the cumulative sales_usd and bottles_sold values for respective vendors.

In PostgreSQL, like before GROUP BY vendor is used to aggregate and HAVING clause is used to filter results of GROUP BY. In SQL, WHERE clause is used to filter rows before they are grouped and aggregated, but the HAVING clause is used to filter the results after they are being grouped.

Query 6: Stores that are doing well over the average

MongoDB:

```
avg_bottles_sold = ls.aggregate([
   {"$group": {"_id": None,
"avg_bottles_sold": {"$avg":
"$bottles_sold"}}}
]).next()["avg_bottles_sold"]

result = ls.aggregate([
   {
      "$match": {
         "bottles_sold": {"$gt":
avg_bottles_sold}
      }
   },
   {
      "$project": {
         "store_name": 1,
         "address": 1,
         "city": 1,
         "county": 1,
         "bottles_sold": 1
      }
   }
])
```

PostgreSQL:

```
SELECT store_name, address, city, county,
bottles_sold
FROM ls
WHERE bottles_sold > (SELECT
AVG(bottles_sold) FROM ls)
```

In MongoDB, the average sales value is obtained by aggregation techniques mentioned above and is assigned to a variable “avg_bottles_sold”. This variable is then used as a reference to filter inside the following query.

In PostgreSQL, the average sales value is obtained by means of using a subquery which is then used as a reference inside a WHERE clause in the main query to filter out the desired stores.

Query 7: Top selling stores per county

MongoDB:

```
pipeline = [
   {
      "$group": {
         "_id": {
            "county": "$county",
            "store_number": "$store_number"
         },
         "total_sales_usd": {"$sum":
"$sale_usd"}
      }
   },
   {
      "$sort": {
         "_id.county": 1,
         "total_sales_usd": -1
      }
   },
   {
      "$group": {
         "_id": "$_id.county",
```

```
        "max_total_sales": {"$first":
"$total_sales_usd"},
        "store": {"$first":
"$_id.store_number"}
      }
   },
   {
      "$project": {
        "_id": 0,
        "county": "$_id",
        "store_number": "$store",
        "total_sales_usd":
"$max_total_sales"
      }
   },
   {
      "$sort": {
        "total_sales_usd": -1
      }
   }
]

result = list(ls.aggregate(pipeline))
```

PostgreSQL:

```
SELECT store_name, address, city, county,
bottles_sold
FROM ls
WHERE bottles_sold > (SELECT
AVG(bottles_sold) FROM ls)
SELECT store_number, county,
total_sales_usd
FROM (
SELECT store_number, county,
SUM(sale_usd) AS total_sales_usd
FROM ls
GROUP BY store_number, count
) AS store_sales
WHERE
(county, total_sales_usd) IN (
SELECT county, MAX(total_sales_usd) AS
max_sales
FROM (
SELECT county, store_number,
SUM(sale_usd) AS total_sales_usd
FROM ls
GROUP BY county, store_number
) AS county_sales
GROUP BY county
)
ORDER BY total_sales_usd DESC;
```

In MongoDB, sales are grouped by county and store number, and the total sales is calculated for each group. The grouped data is sorted. For each county, the store with highest total sales is taken and the corresponding store number is also taken. The final result is displayed in descending order of maximum total sales across all counties.

The PostgreSQL, in the innermost subquery the total_sales is aggregated for each country and store_number and the middle subquery takes the maximum total_sales. The outer subquery groups the results from the other two queries for total_sales in store-county combination. The main query filters the result to only include the rows where the county and total_sales are in the middle query. The final result is displayed in the descending order.

IV. RESULTS

TABLE I
EXECUTION TIMES OF QUERIES IN THE DATABASE

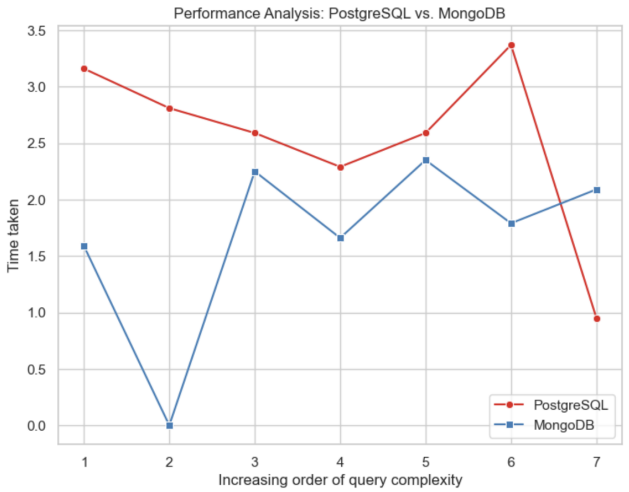
Query	Execution Time		
	PostgreSQL	**MongoDB**	**% difference**
1	3.16	1.59	97.8%
2	2.81	0.006	significantly faster
3	2.59	2.25	15.07%
4	2.29	1.66	37.67%
5	2.59	2.35	10.04%
6	3.37	1.79	87.75%
7	0.95	2.09	54.23% (slower)

THE PERCENTAGE DIFFERENCE:
MEASURES HOW MUCH FASTER IS MONGODB IN COMPARISON WITH POSTGRESQL

$$\text{Percentage Difference} = \left| \frac{\text{MongoDB Time} - \text{PostgreSQL Time}}{\text{MongoDB Time}} \right| \times 100$$

In assessing the execution times of MongoDB and PostgreSQL across distinct queries, notable performance variations were observed. MongoDB showcased a remarkable advantage in the second query, consuming around 15755.37% less time than PostgreSQL. Conversely, PostgreSQL took approximately 15.07% more time than MongoDB in the third query. Subsequent queries demonstrated consistent performance differences, ranging from roughly 10.04% to 87.75%, with PostgreSQL exhibiting longer durations compared to MongoDB. For example, in the sixth query, PostgreSQL recorded an approximately 87.75% longer duration. However, in the seventh query, MongoDB surpassed PostgreSQL by about 54.23%. These execution time variations highlight the contrasting efficiencies of both databases across diverse query complexities, shedding light on their individual strengths and limitations in managing specific tasks.

GRAPH I:



In the graph below we are able to see a comparison model of MongoDB and PostgreSQL. Through this model you can see each query listed on the x-axis in the increasing order of complexity and the time taken to execute by both the databases on the y-axis. This visualization allows us to see how MongoDB and PostgreSQL react to the different levels of query executions. The basic and intermediate queries are executed more efficiently by MongoDB. We can see that PostgreSQL handles the advanced queries with a faster execution time as it has built in features allowing it to be optimal for complex queries.

V. CONCLUSION

In conclusion, the comparative analysis between MongoDB and PostgreSQL highlights nuanced performance attributes in handling varying query complexities. MongoDB often outpaces Postgres due to its denormalized data structure, eliminating the need for resource-intensive table joins. The inherent structure allows for swift operations. Conversely, Postgres, leveraging SQL and an advanced query optimizer, excels in managing intricate queries, showcasing its prowess in handling complex data relationships efficiently. Though JOINS are expensive in PostgreSQL, we can still optimize them by indexing. MongoSB is efficient in handing JSON type data but in recent years there is increased support for JSON in PostgreSQL. Therefore, the choice between MongoDB and PostgreSQL should be contingent upon the specific requirements and intricacies of the anticipated database operations.

VI. REFERENCES

[1] T. Capris, P. Melo, N. M. Garcia, I. M. Pires and E. Zdravevski, "Comparison of SQL and NoSQL databases with different workloads: MongoDB vs MySQL evaluation," 2022 International Conference on Data Analytics for Business and Industry

[2] “Relational Database and NoSQL Inspections using MongoDB and Neo4j on a Big Data Application,” 2022.

[3]“What is MongoDB? — MongoDB Manual,” https://www.mongodb.com/docs/manual/

[4] “PostgreSQL: Documentation,” *www.postgresql.org*. https://www.postgresql.org/docs/

[5] “A Study on Data Input and Output Performance Comparison of MongoDB and PostgreSQL in the Big Data Environment,” https://ieeexplore.ieee.org/document/7433710