Daniel Choi 46773713

Spatial Temporal Data Analysis using Taxi Drivers of Porto, Portugal

Daniel H. Choi – 46773713 || INFS4205 – Advanced Techniques for High Dimensional Data

Abstract—**This document contains analysis of a taxi dataset and possible practical applications of queries such as skyline, spatiotemporal and trajectory similarity. Furthermore, various indexes will be applied to observe and examine their effect on performance of the queries.**

I. INTRODUCTION

THE transportation system is a core functionality of every modern society which enables countless citizens from workers to students to reach their desired destination. Out of these numerous types of transportation, taxis have been one of the most prominent of the public transportation system enabling the customer a direct point to point travelling method. As such, this analysis will focus on the dataset of taxi drivers of Portugal from Kaggle[1].

The dataset consists of 9 columns with 1,703,650 records detailing numerous aspects about the trip. The attributes per record include the following:

- Trip ID
- Call type
- Origin call
- Origin stand
- Taxi ID
- Timestamp
- Day type
- Missing Data
- Poly line

The practical application of the dataset is numerous as the data contains nearly 2 million records with extremely informative attributes. Some of these applications include but are not limited to:

- Most & least efficient taxi ID using distance travelled on average per time.
- Change in taxi usage volume on days prior and on public holidays which would enable the company to allocate more or less drivers.
- The most & least used taxi stand to identify popular and redundant taxi stands.
- The customer which utilizes the taxi service the most often. The company to give them a voucher as a token of appreciation.

This analysis will focus and revolve around the following 3 practical applications in their respective order:

1. Finding the taxi ID and the summation of all their trips which begins or ends in a specified area. This enables the taxi company to reward drivers who have contributed and dedicated the most effort to the company thus enticing other drivers to compete.
2. Find the start and end point of repetitive trips which were **not** made at a taxi stand. This allows the taxi company to create more revenue by placing more taxis or a taxi stand at the given locations. Thus, increasing ease of access to customers
3. Find the 3 closest taxi stands to 3 given locations. If the taxi company decides to develop an app for users to easily find taxis, they can use this skyline query to find the closest taxi stand they can approach to use a taxi.

It is imperative to build efficient index-based queries as it drastically reduces the execution time for the query which has critical benefits. These benefits include but are not limited to:

- Better user experience when using mobile phone applications.
- Drastically faster query execution times.
- Efficient filter application.
- Increased scalability.

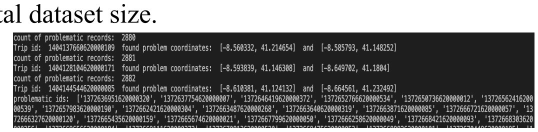
II. METHODOLOGY

The pre-processing and cleaning of data was completed through the combination of PostgreSQL and python. The taxi dataset was extremely noisy and consisted of thousands of data points which were inadequate for use.

Python File 'INFS_clean_data.py'

This python file contains 1 main functionality, which is to identify if any of the polyline data is inconsistent. In this case, polylines are deemed to be inconsistent if the longitude or the latitude values change by a threshold of 0.05. As each data point represents 15 seconds of travel time, it will be physically impossible for any taxi to travel the distance therefore it is ruled as inconsistent and dirty data. The python file stores the trip IDs in an array which enables the trivial use of the "*DELETE*" keyword to remove all effected arrays.

The python file will print out the ID, and the 2 points which exceeded the provided threshold. A total of 2882 records were found to have exceeded the threshold which is about 0.17% of the total dataset size.



Daniel Choi 46773713

Fig. 1. Sample output of the python file.

PostgreSQL – Pre-processing

There were multiple problems with the original dataset which were solved using PostgreSQL which include:

- Original dataset used empty character in place of *NULL*.
- Poly line was given in the format of *TEXT*.
- Certain records contained longitude and latitude within the polyline column.

Improvements & Innovations

All the errors above were addressed the SQL code provided in the SQL file and by creating a staging table. Furthmore, additional columns "start_point" and "end_point" had to be created as PostgreSQL refused to utilize the created GIST indexes when the query contained *ST_StartPoint()* and *ST_EndPoint()* within *ST_Intersects()*. The code to alter the table and add the column has been included in the SQL file code before the first query. Similarly, to use the GIN index on the "call_type" column, an extra "call_type_str" column was created where the type was changed from CHAR(1) to VARCHAR(1). This is because the extension *pg_trgm* does not allow GIN indexes with CHAR data types but allows it for VARCHAR data types.

PostgreSQL – Minor details and intricacies

PostgreSQL will refuse to utilize certain indexes even when it is faster and has less cost in certain situations. These situations vary widely and can include factors such as subqueries and functions being called within another function. As such **ALL** queries had to restructured to not use subqueries and the command "SET enable_seqscan = OFF;" was implemented for certain test cases to ensure the desired indexes would be used.

IMPORTANT NOTE: Indexes were created in a separate query first for fair testing as certain indexes would cause the subsequent query execution time to be increased or sometimes even doubled.

Query 1

Description: The taxi company decides to award a driver from each section of the city who has achieved highest total trip time.

Conditions:

- The trip will only count as being associated with an area if it started or ended in that area otherwise, taxi drivers can drive through every single district to receive multiple rewards.

Indexes implemented:

1. Sequential Scan
2. BRIN index on "timestamp"
3. GIST index on "line geometry"

Query 2

Description: Find the most common trip taken by customers in a time period which enables the taxi company to place a taxi stand.

Conditions:

- The trip must be ordered on the street.
- 2 trips are considered to be the same when the HausdorffDistance between the 2 trips are smaller than a given threshold.
- The start & end points must be within another given threshold.
- Length of the trip must be between 0.01 and 1.
    - Trips over the length of 1 lead out of Porto therefore is out of the taxi company's jurisdiction.

Indexes implemented:

1. Sequential Scan
2. HASH index on "call_type" with b-tree on timestamp
3. GIN index on "call_type" with b-tree on timestamp

Query 3

Description: The location of the k closest taxi call stand to 3 given coordinates. By entering population destinations into the skyline query, the results will show which call stands lie equidistant from each of those 3 popular destinations. At these taxi stands, more taxis can be stationed as it is more likely that there will be numerous customers desiring to use a taxi to travel to those popular destinations.

For example, a taxi stand between 2 airports and a popular restaurant is likely overwhelmed as it must deal with customers wanting to travel to the airports and the customers wanting to travel to the restaurant. Therefore, if the taxi company was to station more taxis at this stand, there will be less wait time for the customers and more revenue will be generated.

Indexes implemented:

1. Sequential Scan
2. HASH index on "call_type"
3. BRIN index on "call_type"

III. EXPERIMENTAL RESULTS & ANALYSIS

The results and analysis will be split up per query.

Query 1 – Spatiotemporal query

Index	Test Case	Execution time (ms)	Index cost
Sequential Scan	1	8369.379	18754552.47
	2	7779.159	18754552.47
	3	8255.905	18754552.47
BRIN	1	462.321	154.60
	2	430.324	154.60
	3	3296.816	327.62
GIST	1	4535.316	11.14 *
	2	5608.963	11.14 *
	3	488.399	4030.68 *

* Index cost of *gist_start_geom* and *gist_end_geom* are combined

Fig. 2. Query 1: Raw data table

Index	Average Time	Average cost
Sequential Scan	8134.81433	18754552.5
BRIN	1396.487	212.273333
GIST	3544.226	1350.98667

Fig. 2.1. Query 1: Average time and cost per index

For further processing, implemented *ST_Area()* to calculate the respective area of the polygon square metres.

Test Case	Average Time	Average cost	Area (m^2)
1	4455.672	6251572.737	0.00540
2	4606.148667	6251572.737	0.00555
3	4013.706667	6252970.257	0.02170

Fig. 2.2. Query 1: Area, average cost and time per trial

* Area has been rounded to 5 decimal points for ease of readability

As shown in *figure 2.2*, the overall area covered by the spatial query is quintuple in test case 3 comparative to the other test cases. This was done intentionally to examine and analyse the utilisation of indexes and their respective performance as the area of the spatiotemporal query increased for scalability.

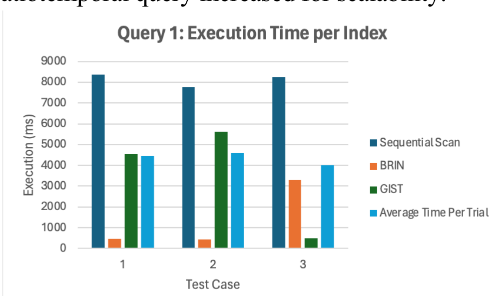


Fig. 2.3. Query 1: Execution time per index

Query 1: Time analysis

There are several distinctive trends that can be noticed by observing *figure 2.3*. The most obvious is that sequential scan is consistently the slowest method regardless of the area given to the query. However, more interesting observation is that as the area of the polygon provided to the spaciotemporal query increased, the execution time of the GIST index was vastly superior to that of the other indexes. The BRIN index has the opposite effect where the execution time drastically increased as the area of the polygon increased. Therefore, with large polygons, GIST indexes are the most efficient whereas for smaller areas can implement the BRIN index instead for execution time. This effect is caused as, the number of spatial operation drastically increase as the area of the polygon increases, and as spatial operations heavily utilise the GIST index, the overall query time is lower with GIST.

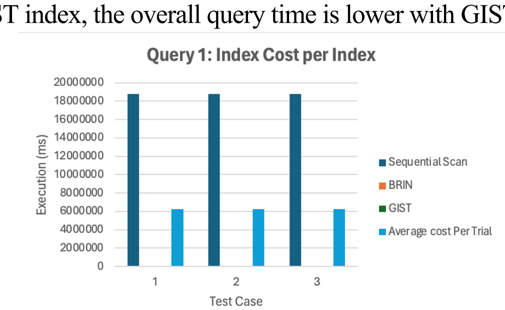


Fig. 2.3. Query 1: Index cost per index

Query 1: Cost analysis

The index cost of the indexes displays a similar trend where sequential scan has a substantially higher cost than all other indexes. When viewing *figure 2.1*, it is evident that the average cost of the BRIN index is lower as GIST indexes are more complex dealing with multiple dimensions.

Query 2 – Trajectory similarity

Index	Test Case	Execution time (ms)	Index cost
Sequential Scan	1	554261.223	20086170.66
	2	675538.017	20086170.66
	3	576047.435	20086170.66
HASH + b-tree	1	522511.287	38868.18
	2	633691.420	42735.88
	3	553323.703	42704.14
GIN + b-tree	1	548494.211	18589.38
	2	667055.658	18589.38
	3	558880.572	18557.68

Fig. 3. Query 2: Raw data table

Query 2 took a slightly differed approached to the other queries, as to investigate if a combination of indexes would aid to increase the performance and memory overhead. Furthermore, query 2 was significantly more complex due the numerous conditions it held thus making it more computationally expensive.

Index	Average Time Per Trial	Average cost Per Trial
Sequential Scan	541755.574	6714542.74
HASH + b-tree	658761.698	6715831.97
GIN + b-tree	562750.57	6715810.83

Fig. 3.1 Query 2: Processed data table

Fig. 3.2. Query 2: Execution time per index

Query 2: Initial analysis

At the first observation of *figure 3.2* and *figure 3.3,* it seems that all indexes performed at a similar performance, however upon closer inspection, the rate at which the y axis increases is several magnitudes higher than the other graphs. Therefore, the differences between the indexes are still large.

Query 2: Time analysis

As seen in *figure 3.2,* the HASH index was able to consistently outperform the average time whereas the GIN index was similar to the average time on all test cases. As usual the sequential scan performed the worst by achieving the highest time in all three test cases.

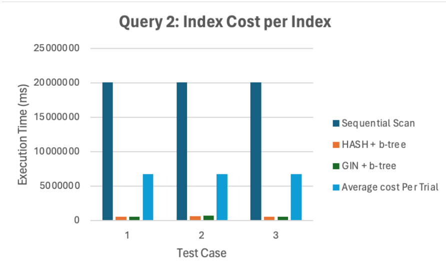


Fig. 3.2. Query 2: Index Cost per index

Query 2: Cost analysis

As displayed in *figure 3.2,* both HASH and GIN index in combination with a b-tree was able to drastically outperform the sequential scan and the average cost per trial. The sequential scan had an extremely noticeable and large disadvantage to all other indexes as shown by the extremely high bar. In addition, the b-tree index seems to be efficient as the memory use is very minimal whilst improving execution time.

Query 3 – Skyline query

Index	Test Case	Execution time (ms)	Index cost
Sequential Scan	1	10734.948	13996379.87
	2	11953.208	14029422.93
	3	11212.041	14029422.93
HASH	1	3160.331	26088.08
	2	2987.762	26088.08
	3	2727.218	26088.08
BRIN	1	10608.376	327.62
	2	10545.659	327.62
	3	11576.373	327.62

Fig. 4. Query 3: Raw data table

Index	Average Time	Average cost
Sequential Scan	11300.06567	14018408.58
HASH	2958.437	26088.08
BRIN	10910.136	327.62

Fig. 4.1. Query 3: Processed data table

Fig. 4.2. Query 3: Execution time per index

Query 3: Time analysis

As showcased in *figure 4.2*, the execution time for the HASH index was superior to all other indexes and was the only index to be below the average execution time per trial for every test case. Furthermore, the BRIN index outperformed the sequential scan on all test cases except the last test case. Since the hash index stores a 32 bit hash code derived from the value [3], the equal operator will be extremely quick as all it must do is check if the two hash codes are identical.

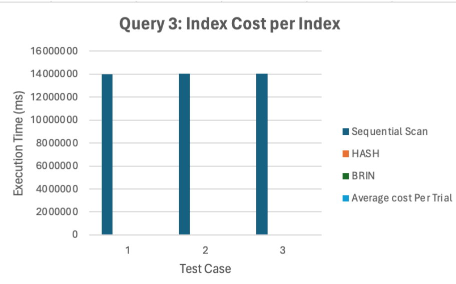


Fig. 4.3. Query 3: Index cost per index

Query 3: Cost analysis

As the bar for the sequential scan is the only visible bar in *figure 4.3,* it is evident that the memory cost of sequential scan is substantial larger than the HASH and BRIN indexes. Upon further examination, we notice that the cost of the BRIN index is the lowest as shown in *figure 4.1* however the trade-off between the execution time and cost is too great for BRIN to be considered as an effective index. The BRIN index has the smallest cost comparative to HASH as it only stores the summary values of the given block range therefore it minimizes memory usage. The HASH index is also extremely small in size as it only stores the 32-bit hash values [3] and the columns only contain 3 separate values for "*call_type*" which are "A", "B" and "C".

IV. Conclusion

In conclusion, the best index for spatiotemporal queries is BRIN for when the area is small and GIST for when the area is large. For trajectory similarity, the combination of HASH and a b-tree was the most effective in terms of execution time however the GIN and b-tree index was superior in terms of memory usage. Lastly, for skyline queries, the HASH index was the most efficient in terms of execution time. The BRIN index was more efficient in terms of memory usage however it was drastically slower to the point where it would not be scalable for bigger skyline queries therefore HASH index would still be superior.

References

[1] *Cross, C. (2018). Taxi Trajectory Data. Www.kaggle.com. https://www.kaggle.com/datasets/crailtap/taxi-trajectory/data*

[2] *Gupta, R. (2021, March 24). A guide to using Postgres indexes. The Quest Blog. https://blog.quest.com/a-guide-to-using-postgres-indexes/*

[3] *Yu, A. (2022, February 10). 11.2. Index Types. PostgreSQL Documentation. https://www.postgresql.org/docs/current/indexes-types.html*