Comparative Performance Analysis of Algorithms and Indexing Techniques for Spatial-Temporal Linestring Data

Nguyen Van Quoc Chuong, 47787810, ITEE, The University of Queensland, Australia

***Abstract*—This research paper aims to compare the performance of various algorithms and indexing techniques, including R-tree, kd-tree, ball-tree and quad-tree, in the context of spatial-temporal linestring data. The study focuses on evaluating the algorithms based on two key metrics: creation time, execution time. The postGIS query output is employed as the benchmark for the evaluation.**

The research encompasses several query scenarios, including: (a) retrieving all data linestrings within a specified rectangular area and time window, (b) identifying data points within a certain distance from a trajectory occurring on the same day, (c) finding the k nearest neighbors of a given trajectory for a specific date, (d) identifying the shortest trajectory between two given data points, and (e) finding the trajectory most similar to a given trajectory.

The findings of this research provide valuable insights into the efficiency and effectiveness of different algorithms and indexing techniques for spatial-temporal linestring data. These results assist researchers, practitioners, and database administrators in selecting suitable methods for specific query requirements. Additionally, the study contributes to the advancement of spatial-temporal data management by offering empirical evidence on the performance characteristics of these algorithms and indexing techniques.

Keywords: spatial-temporal data, linestrings, R-tree, kd-tree, ball-tree, quad-tree, performance comparison, polyline.

I. Introduction

Spatial-temporal data, which combines spatial and temporal attributes, plays a crucial role in various domains, such as transportation and urban planning. Managing and querying such data efficiently is essential for extracting valuable insights and supporting decision-making processes. In this context, algorithms and indexing techniques are pivotal components for optimizing query performance.

This research paper focuses on the comparative analysis of algorithms and indexing techniques for spatial-temporal linestring data. Linestrings represent continuous trajectories or paths, capturing the movement of objects or entities over time. The objective of this study is to evaluate the performance of different approaches when applied to spatial-temporal linestring data, specifically considering creation time, execution time as key metrics.

The research explores several query scenarios commonly encountered in spatial-temporal data analysis. These queries include retrieving data points within a specified rectangular area and time window, identifying relevant data points based on distance from a trajectory occurring on the same day, finding the k nearest neighbors of a given trajectory for a specific date, identifying the shortest trajectory between two points, and finding the trajectory most similar to a given trajectory.

The findings of this research will contribute to the field of spatial-temporal data management by providing empirical evidence on the efficiency and effectiveness of various algorithms and indexing techniques for spatial-temporal linestring data. The insights gained from this study will aid in the selection and implementation of appropriate methods for specific query requirements, ultimately enhancing the overall performance and usability of spatial-temporal data systems.

The subsequent sections of this paper present the methodology employed, discuss the experimental setup and results, and provide a comprehensive analysis and discussion of the findings. Finally, the conclusions drawn from this study are presented, followed by suggestions for future research directions in the domain of spatial-temporal data management.

II. Preprocessing

In the preprocessing step, several data cleaning and transformation operations were performed. Firstly, null values were removed by dropping rows where the `missing_data` column was marked as True, and the `missing_data` column was removed from the dataframe. Additionally, rows with empty `polyline` values were also eliminated.

Next, the total distance of polylines was calculated using the great circle formula. Each pair of coordinates in a linestring was traversed, and the distance between them was calculated. If the total distance was found to be less than 0.4 km, the linestring was dropped from the dataset to reduce its size while considering the relative information content.

To prepare the linestring data for further analysis, the `polyline` column's values were converted from string format to the LineString format using the Shapely Python library. Additionally, the start and end points of each linestring were extracted and stored in the dataframe.

Further data refinement involved removing records where the start point was the same as the end point, indicating no movement. Here, we assume that the taxi moves between two consecutive coordinate pairs at a constant speed within 15 seconds, unusual distance changes between the coordinate pairs will be considered invalid. Invalid coordinate pairs were also detected using two filters. The first filter identified unusual distances between coordinate pairs within a 15-second timeframe. Outlier distances found using interquartile range (IQR),

assumed to follow a Gaussian distribution, were considered invalid if they also exceeded a threshold of 0.51 (representing a maximum velocity of 120 km/h). This criterion aimed to identify cases where the vehicle potentially exceeded the speed limit in real life. In the case of a taxi traveling over 120km/h over the whole distance (but not the outlier), well, in this case we still consider it valid.

The second filter addressed the issue of lost GPS connection when a vehicle entered and exited the Amadas tunnel in Porto, Portugal. By manually collecting 40 linestrings passing through the tunnel, the mean distance during tunnel traversal was calculated and used as a threshold (0.93 km) Linestrings with distances exceeding this threshold were considered invalid trips.

Finally, to accommodate the computational limitations of the hardware, the dataset was reduced from 1.9 million rows to 1 million rows before importing it into the PostgreSQL database.

The preprocessing steps ensure data quality and relevance, facilitating efficient analysis of the spatial-temporal linestring data in the subsequent stages of the research.

III. Query tasks

A. Query task 1: Find all taxi trips in a given rectangular area and within a certain time (range) window

To retrieve all taxi trips that occurred within a specified rectangular area and time range, three indexing techniques were employed: R-tree, KD-tree, and Quad-tree.

In the R-tree creation process, each linestring in the dataset was associated with a bounding box. Initially, the minimum perimeter rectangular bounding box for each linestring was determined using the `minimum_rotated_rectangle` function from the Shapely library in Python. However, due to unsupported parameters, this specific type of bounding box could not be inserted into the R-tree structure. Nonetheless, it is worth noting that utilizing the minimum perimeter bounding box has the potential to reduce overlap in the R-tree, leading to faster query performance.

For the KD-tree creation, a 3D bounding box was generated for each linestring, incorporating the x, y, and timestamp dimensions. These bounding boxes were then added to the KD-tree structure, enabling efficient spatial and temporal searching.

Similarly, the Quad-tree creation involved calculating the entire bounding box encompassing all linestrings in the dataset. This bounding box, representing the maximum extent of the spatial data, was utilized as a parameter in the Pyqtree library to construct a Quad-tree. Instead of using a 3D bounding box, a 2D point (excluding the timestamp) was employed to represent each linestring in the Quad-tree.

To execute the query, the `intersection()` function was applied to both the R-tree and the relevant tree structure (KD-tree or Quad-tree) to identify intersecting trips within the specified rectangular area. This step efficiently narrowed down the search space and provided a set of potential trip IDs.

Subsequently, a linear scan technique was employed to validate that the taxi trips actually intersected with the rectangular area within the designated time range. For each potential trip ID, a linear scan was performed on the associated linestring's coordinate pairs. By checking if any point fell within the rectangular area and time range, the true intersecting trips were identified and selected.

By combining the results obtained from the R-tree, KD-tree, and Quad-tree, and utilizing the linear scan technique, all taxi trips meeting the defined spatial-temporal criteria could be efficiently retrieved.

B. Query task 2:Find k nearest neighbours of a given point for a given time range

To find the k nearest neighbors (start data points/trip IDs) of a given point within a specified time range, three indexing techniques were utilized: R-tree, KD-tree, and Ball-tree.

In the R-tree approach, the start points of each trip were inserted into the R-tree structure instead of the entire linestring. Each start point was associated with a 3D bounding box, including the x and y coordinates (representing the start location) and the timestamp. By utilizing the `nearest()` function, the R-tree efficiently identified the nearest start points to the given search coordinates. However, to ensure the validity of the timestamps, a linear scan was performed on the returned trip IDs to filter out the start points within the specified time range.

For the KD-tree and Ball-tree methods, the process was similar. The start points were represented as 2D points (excluding the timestamp) and used as input for the NearestNeighbors class from the `sklearn` library. The KD-tree and Ball-tree were constructed using the 'kd_tree' and 'ball_tree' methods, respectively. To retrieve the k nearest neighbors, the `kneighbors()` function was employed, specifying the desired number of neighbors (n_neighbors = 3000 in this case, which may vary depending on the specific scenario). Similarly, a linear scan was conducted on the returned nearest neighbors to filter out the valid start points within the given time range.

It is important to note that the choice of the `n_neighbors` is crucial and depends on the specific requirements of the problem. In this case, a value of 3000 was used to cover the necessary range, but it may differ in other scenarios. Additionally, the linear scan step was necessary to validate the timestamps and ensure that the selected start points fall within the specified time range.

C. Query task 3: Find all data points (start point of other linestrings) within certain distance to a trajectory (start point of the given linestring) emerging on the same day.

In this query task, the goal is to identify all data points (start points of other linestrings) that lie within a specified distance of a given trajectory (start point of a specific linestring) that emerges on the same day. This query task assumes that each taxi trip is considered independently, without any interference or stops to pick up additional passengers during transit.

Two approaches were employed to address this query task:

Approach 1: K-Nearest Neighbors within a Distance (Radius): Initially, a value for the distance radius (r) was chosen, which is large enough to cover the relevant data points.

The K-nearest neighbors within this radius were identified using the KD-tree and Ball-tree data structures with the `radius_query` function. Subsequently, a linear scan was conducted on the returned points to filter based on both distance and timestamp (ensuring that the points were from the same day). This approach offers an efficient way to find nearby points within a specified radius.

Approach 2: Region Search based on Bounding Box: In this approach, a region (bounding box) was determined by calculating the coordinates from the target point and adding/subtracting a specific target distance. The R-tree data structure was employed to find all points within this bounding box, representing the potential data points within the desired distance range. Similar to Approach 1, a linear scan was performed to filter based on both distance and timestamp. It is worth noting that this approach is showcased as a demonstration, and the same query can be implemented using other indexing techniques such as KD-tree.

To compare the performance of these approaches, the execution times of Approach 1 (using KD-tree and Ball-tree) and an example of Approach 2 (using R-tree) were measured. Other variations of Approach 2 were not included in this task. The performance analysis for region search was presented in Task 1.

The implementations for these approaches follow the same structure as described in Task 1, with the additional step of performing a linear search to verify that the points correspond to the same day as the given trajectory and to check the distance between the points and the given point.

D. Query task 4: Find the trajectory that is shortest (NOT fastest) from given data point to another.

The goal of this query task is to find the shortest road between two given points on the map, assuming that the shortest road can be considered the fastest route. It is important to note that the data used in this research is not normalized, meaning that the GPS coordinates collected every 15 seconds may not represent equal distances due to various factors such as traffic conditions or stops along the road. Consequently, the focus is on determining the shortest road based on the total distance traveled in each taxi trip.

The approach to solving this query task involves the following steps:

1. Find all the start points and end points within a rectangular area that is 100 meters from the given points. By querying the dataset based on this region, the associated trip IDs of the taxi trips passing through these regions are obtained.

2. Utilize linear search techniques to find the minimum distance between the start and end points. This involves traversing all the points in the linestring of each trip and calculating the distances (foward distance and backward distance) for each pair of consecutive points. By taking the minimum of these distances, the minimum distance from the first point appeared in region 1 and the first point appeared in region 2 is determined. The taxi trip with the minimum distance is considered the shortest road.

It is important to note that the focus of this task is on finding the shortest road rather than the fastest path. The assumption is that if the shortest road is determined, it can be implicitly assumed to be the fastest, despite the lack of normalization in the data.

E. Query task 5: Find trajectory that is most similar to a given trajectory.

The objective of this query task is to identify the trajectory in the dataset that is most similar to a given trajectory. The similarity between two linestrings is defined based on their start and end points being located in the same region. To simplify the implementation, this query task can be combined with Task 4.

The approach for this query task involves the following steps:

1. Given two start and end points, calculate the bounding box that covers an area of approximately 100 meters around these points.

2. Apply intersection operations on the R-tree index structure, similar to the approach used in Task 1, to identify the trip IDs of the taxi trips that pass through the corresponding rectangular regions.

3. Utilize linear search techniques to validate that the taxis actually traverse the identified rectangular regions.

In addition, there are many other complicated methods to check if two linestrings are similar (eg. Synchronous Euclidean Distance (SED) or Locality In-between Polylines Distance (LIP), etc), but within the scope of the article, these methods will not be mentioned.

IV. Performance comparison

The performance evaluation of the different indexing techniques reveals interesting insights.

Regarding the creation time, the R-tree exhibits the slowest performance compared to the other techniques. On the other hand, both the Ball-tree and KD-tree demonstrate relatively faster creation times.

When considering query times, the Ball-tree stands out as the fastest indexing technique, providing efficient search capabilities. In contrast, the linear scan approach consistently exhibits the slowest query times among all the indexing techniques.

These findings suggest that the Ball-tree indexing technique offers a favorable trade-off between creation time and query performance, making it a promising choice for spatial-temporal linestring data analysis.

A. Query 1

	R-tree	Quad-tree	KD-tree	Linear scan
Creation time	00.000997	00.917516	00.003987	-
Query time	00.162609	01.227216	00.103820	23.558182

	R-tree	KD-tree	Ball-tree
Creation time	01:55.730536	00:02.547452	00:02.308258
Query time	00.006981	00.006981	00.005983

	Linear scan
Creation time	-
Query time	00.427306

B. Query 2

C. Query 3

The linear scan approach (implemented in Python) is significantly slower in this scenario due to the lack of indexing techniques compared to the database. R-tree, while offering spatial indexing capabilities, proves to be less efficient for finding intersections, resulting in longer processing times. On the other hand, both KD-tree and Ball-tree exhibit favorable performance. However, it is worth noting that the initial selection of the radius "r" remains a crucial factor to consider for optimizing the efficiency of these tree-based approaches.

	R-tree	KD-tree	Ball-tree
Creation time	1:14.505482	02.828413	02.383334
Query time	05.804482	00.061834	00.048869

D. Query 4, 5

The evaluation of query 4 and 5 in this report primarily focuses on the performance of the `filter()` function. The creation of indices and finding the points within a rectangle follows the same approach as in task 1. To verify if the taxis actually pass through the specified searching bounding boxes, a linear scan method is employed, similar to task 1. Considering the dataset size of over 49 million points, it is challenging to achieve higher query performance without compromising data storage capacity. As a result, the worst-case time complexity for this task is O(n^2).

	with R-tree	Linear scan only
Creation time	00.053856	-
Query time	2:25.413369	31:35.530149

V. Conclusion

In this research, we explored the performance of different spatial indexing methods for analyzing taxi trip data. Our study focused on three popular indexing techniques: R-tree, Quad-tree, Ball-tree and KD-tree, along with a linear scan approach for comparison.

Through our experiments, we found that the choice of spatial indexing method significantly impacts both the creation time and query time in analyzing large-scale taxi trip datasets. The R-tree demonstrated slower creation times compared to Quad-tree and KD-tree, but it exhibited competitive query times, making it a suitable choice for efficient spatial queries.

The KD-tree and Ball-tree showed impressive performance in both creation time and query time, making it a promising choice for efficient spatial indexing. Its ability to partition the data into balanced partitions along each dimension contributed to its fast query processing capabilities.

In contrast, the linear scan approach exhibited significantly slower query times compared to the spatial indexing methods. While it may be suitable for smaller datasets or scenarios where query speed is not critical, it is not scalable for large-scale analyses.

Based on our findings, we recommend the adoption of spatial indexing methods, such as R-tree, Ball-tree, or KD-tree, for efficient querying and analysis of taxi trip data. The choice of the indexing method should be based on the specific requirements of the application, considering factors such as dataset size, query frequency, and creation time constraints.

Future research can focus on exploring advanced indexing techniques and optimizations to further enhance the performance of spatial queries on taxi trip data. Additionally, investigating the scalability and performance of these indexing methods on different types of spatial datasets and real-time analysis scenarios would provide valuable insights.

Overall, this research contributes to the understanding of spatial indexing methods and their applicability in the analysis of taxi trip data. By leveraging efficient indexing techniques, researchers and practitioners can gain valuable insights from large-scale taxi trip datasets, leading to improved transportation planning, route optimization, and urban mobility management.

Acknowledgment

We would like to acknowledge our academic advisors (Prof. Helen and Prof. Yadan) and mentors for their guidance and support throughout the research process. Their expertise and valuable feedback greatly contributed to the development and refinement of our methodologies and results.

References

[1] Taxi-trajectory dataset: ECML/PKDD 15: Taxi Trip Time Prediction (II) Competition
[2] https://en.wikipedia.org/wiki/Great-circle_distance
[3] https://en.wikipedia.org/wiki/Haversine_formula
[4] https://shapely.readthedocs.io/en/stable/manual.html