Implementation Report

Jinming Ge

November 24, 2025

1 Introduction

Spatial queries are fundamental operations in geographic information systems (GIS) and spatial databases. In Assignment 1, two spatial query methods were implemented: *point-in-polygon* queries and *polygon-in-polygon* queries.

However, without proper indexing, spatial queries become computationally expensive, particularly when dealing with large-scale geospatial datasets. This research proposes to compare the efficiency of different spatial indexing methods for the aforementioned query operations. Specifically, four prominent indexing techniques will be evaluated:

- **Z-order curve:** A space-filling curve, also known as Morton Order, that linearly maps multi-dimensional spatial data into a one-dimensional sequence by interleaving the binary representations of coordinate values. By clustering spatially proximal points into close intervals in the linear sequence, it allows complex multi-dimensional range queries to be reduced to efficient one-dimensional operations like binary search.
- **R-tree:** A balanced, dynamic hierarchical structure that organizes spatial objects by grouping them into Minimum Bounding Rectangles (MBRs). Unlike static grids, the R-tree adapts to the distribution of the data, where each internal node represents the aggregate spatial extent of its children, facilitating efficient querying for both point and region data in dynamic environments.
- **R*-tree:** An optimized variant of the R-tree that employs rigorous heuristic strategies during node splitting and insertion. It explicitly aims to minimize not just area, but also margin (perimeter) and overlap between MBRs. Furthermore, it introduces the concept of *forced reinsertion*, which dynamically reorganizes the tree structure to achieve a more compact index, significantly improving query performance at the cost of higher construction latency.
- **Quad-tree:** A hierarchical spatial data structure that recursively partitions the two-dimensional domain into four quadrants (nodes), facilitating the indexing of spatial objects by decomposing the space into successively smaller regions.

The research [1]empirically evaluate these indexing methods in terms of query response time, index construction overhead, and memory consumption.

2 Formal Definition of Spatial Queries

In this section, we provide the formal definitions for the implemented spatial queries. Let $\mathcal{D}$ represent the spatial domain (e.g., $\mathbb{R}^2$). We define the spatial objects and the containment predicate based on the functions provided in the system implementation (see Listing 1).

```
MAKE_POINT(latitude, longitude)
    -- Constructs a geometric point object from coordinates

MAKE_POLYGON(vertex_table_name, id_column, id_value)
    -- Constructs a polygon object by fetching and ordering
    -- vertices from the specified vertex table

ST_Contains(geometry1, geometry2)
    -- Returns TRUE if geometry1 spatially contains geometry2

ST_Distance(geometry1, geometry2)
    -- Computes the minimum distance between two geometries
```

Listing 1: Spatial Operation Definitions

[1]This project is fully open-sourced at https://github.com/gjskywalker/COMP6311G_Assignments.

2.1 Preliminaries and Notations

1. **Point (p):** A zero-dimensional geometric entity defined by a coordinate pair. In our system, a point is instantiated via $\texttt{MAKE\_POINT}(lat, lon)$.
2. **Polygon (s):** A two-dimensional geometric region defined by a sequence of ordered vertices forming a closed ring. A polygon is instantiated via the `MAKE_POLYGON` function, which aggregates vertex data.
3. **Containment Predicate (θ):** We utilize the boolean function `ST_Contains` to evaluate topological inclusion.

Mathematically, the containment relationship is defined as:

$$\texttt{ST\_Contains}(g_1, g_2) \iff g_2 \subseteq g_1$$

This condition holds true if and only if no points of geometry g_2 lie in the exterior of g_1, and at least one point of the interior of g_2 lies in the interior of g_1.

2.2 Query 1: Point-to-Polygon Query ($Q_{pt2poly}$)

Definition: The Point-to-Polygon query retrieves all pairs consisting of a specific point (or set of points) and the polygons that strictly spatial contain them.

Formal Formulation: Let $P = \{p_1, p_2, \dots, p_n\}$ be a dataset of points and $S = \{s_1, s_2, \dots, s_m\}$ be a dataset of polygons. The query returns a result set R_{pt} defined as:

$$R_{pt} = \{(p, s) \in P \times S \mid \texttt{ST\_Contains}(s, p) = \text{TRUE}\}$$

Implementation Logic: For a specific point defined by coordinates (lat, lon) and a target polygon set constructed from relation T_{poly} with ID column col_{id}, the query logic equates to:

$$\texttt{ST\_Contains}\Big(\texttt{MAKE\_POLYGON}(T_{poly}, col_{id}, val), \texttt{MAKE\_POINT}(lat, lon)\Big) \equiv \text{TRUE}$$

2.3 Query 2: Polygon-in-Polygon Query ($Q_{poly2poly}$)

Definition: The Polygon-in-Polygon query determines the topological partial ordering between two sets of polygons. It identifies if a "subject" geometry (e.g., a building footprint) is entirely contained within a "target" geometry (e.g., a city district).

Formal Formulation: Let S_{inner} be the set of candidate inner polygons and S_{outer} be the set of candidate outer container polygons. The query returns a result set R_{poly} representing the subset of the Cartesian product satisfying inclusion:

$$R_{poly} = \{(s_{in}, s_{out}) \in S_{inner} \times S_{outer} \mid \texttt{ST\_Contains}(s_{out}, s_{in}) = \text{TRUE}\}$$

Implementation Logic: Let the inner polygon be constructed from table T_A and the outer polygon from table T_B. The query condition is satisfied if:

$$\texttt{ST\_Contains}(\text{poly}_{outer}, \text{poly}_{inner}) \equiv \text{TRUE}$$

Where:

- $\text{poly}_{outer} = \texttt{MAKE\_POLYGON}(T_B, id_B, val_B)$
- $\text{poly}_{inner} = \texttt{MAKE\_POLYGON}(T_A, id_A, val_A)$

3 Implementation

3.1 Experimental Environment

The implementation and performance benchmarks were conducted on a specific hardware and software configuration to ensure reproducibility. The environment consists of:

- **Spatial Data Management:** The system relies on ipython 8.15.0 and the `GeoPandas 1.1.1` library for data management. This stack handles the geometric instantiation and topological predicates (such as `ST_Contains`) described in Listing 1, operating effectively in-memory without the overhead of a client-server database.
- **Testing Platform:** All experiments are executed on a workstation running Ubuntu 22.04.4 LTS (Jammy Jellyfish). The system is powered by a 13th Gen Intel® Core™ i9-13900 CPU (x86_64 architecture), equipped with 24 cores (32 threads) to support intensive spatial computations.

3.2 Data Acquisition and Characteristics

The experimental datasets are curated to align with the spatial query requirements defined in Assignment 1. Data is acquired from authoritative government and scientific repositories to ensure topological validity.

- **Flora and Fauna Sightings (Point Data):** Sourced from the Atlas of Living Australia (ALA). To evaluate performance across different data magnitudes, we select two distinct species: a high-volume dataset containing 1,454,283 sightings of *Legless Lizards* and a moderate-volume dataset containing 29,656 sightings of the *Platypus*.
- **Environmental Zones (Polygon Data):** Sourced from the Queensland Spatial Catalogue. This includes two key layers. **Protected Areas:** A compilation of 2,214 state and national forest parks (Original CRS: EPSG:7844). And **Wetlands:** The *Directory of Important Wetlands* dataset, comprising 335 aggregate wetland regions (Original CRS: EPSG:4283).

To ensure the accuracy of metric computations (specifically distance and containment), all datasets are re-projected from their original geographic coordinate systems (degrees) to **EPSG:3577 (GDA94 / Australian Albers)** prior to experimentation. This transformation facilitates precise Euclidean operations in meters.

3.3 Experimental Design

The evaluation framework is structured into three distinct phases to assess the critical performance characteristics of each spatial index. **Phase 1 (Index Construction)** benchmarks the initialization costs, specifically quantifying construction latency, memory footprint, and structural depth. **Phase 2 (Query Performance)** subjects the indexes to rigorous Point-in-Polygon (PiP) and Polygon-in-Polygon containment workloads, measuring average response times and throughput across varying spatial distributions. Finally, **Phase 3 (Comparative Analysis)** evaluates longer-term scalability and robustness, analyzing performance degradation relative to dataset growth, query selectivity, and update frequency. What's more, the following algorithm is designed to minimize nearest-neighbor search (question 3) overhead.

Algorithm 1: Iterative k-NN Search

```
Input: Query Point $P$, Packed Index $\mathcal{T}$, Max Radius $R_{max}$
Output: Nearest Object $O_{near}$, Minimum Distance $d_{min}$
1  $r \leftarrow 10,000$ ;                                   // Initialize search radius (10km)
2  $d_{min} \leftarrow \infty$;
3  $O_{near} \leftarrow$ null;
4  while $r \leq R_{max}$ do
       // Construct Search Bounding Box
5      $B \leftarrow$ MakeBox$(P, r)$;
       // Execute Range Query on Data Structure
6      $\mathcal{C} \leftarrow \mathcal{T}$.Query$(B)$ ;                 // Set of candidate objects
7      if $\mathcal{C} \neq \emptyset$ then
8          foreach $O_i \in \mathcal{C}$ do
9              $d_{curr} \leftarrow$ ST_Distance$(P, O_i)$;
10             if $d_{curr} < d_{min}$ then
11                 $d_{min} \leftarrow d_{curr}$;
12                 $O_{near} \leftarrow O_i$;
13             end
14         end
           // Termination Check: Is the nearest object within current safe radius?
15         if $d_{min} < r$ then
16             return $(O_{near}, d_{min})$ ;                    // Global nearest confirmed
17         end
18     end
       // Expand search space and retry
19     $r \leftarrow r \times 2$;
20 end
21 return $(O_{near}, d_{min})$;
```

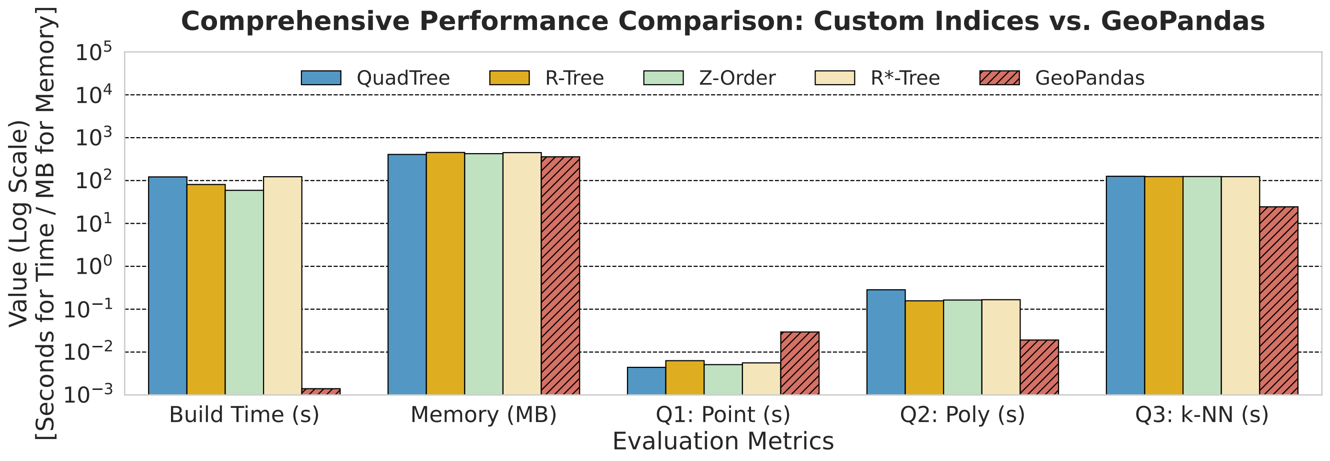

Figure 1: Performance Comparison Between Different Data Structures

4 Experiment Results

4.1 Index Construction and Space Efficiency

The experimental results highlight distinct trade-offs between construction complexity and memory consumption.

- **Construction Latency:** The `Z-Order` demonstrated the most efficient construction time (59.16s) among the custom implementations, outperforming the standard R-Tree (80.91s) and the computationally intensive R*-Tree (122.82s). This validates the efficiency of bulk-loading techniques over iterative insertion. However, `GeoPandas` exhibited negligible construction time (0.0014s), primarily because it wraps the highly optimized C-based `GEOS` STRtree library, which avoids Python interpreter overhead during build time.
- **Memory Footprint:** `GeoPandas` proved to be the most space-efficient (358.21 MB). Among custom structures, the `QuadTree` consumed the least memory (406.28 MB), while the R-Tree variants required approximately 10-15% more space due to the overhead of storing bounding box coordinates and node pointers for the complex hierarchy.

4.2 Query Processing Performance

The query benchmarks reveal the impact of algorithmic complexity and language-level optimizations.

- **Point-in-Polygon (Query 1):** Surprisingly, custom Python implementations outperformed GeoPandas (e.g., QuadTree: 0.0044s vs. GeoPandas: 0.0294s). This anomaly suggests that for simple point containment checks on smaller datasets, the overhead of initializing GeoPandas Series/DataFrames and invoking C-extensions may exceed the cost of a direct Python object traversal.
- **Polygon-in-Polygon (Query 2):** `GeoPandas` (0.0191s) was roughly 8× faster than the best custom implementation (R-Tree: 0.1568s). This illustrates the advantage of vectorized operations and compiled C geometric predicates (STRtree) over Python-based intersection loops.
- **Nearest Neighbor (Query 3):** This query highlighted the most significant performance divergence. `GeoPandas` completed the task in 24.40s, whereas all custom R-Tree variants hovered around 123-125s. While the custom iterative k-NN algorithm is theoretically sound, it suffers from the lack of vectorization. GeoPandas utilizes the `nearest()` spatial index method, which performs the distance calculations in the underlying C GEOS layer, drastically reducing CPU cycles compared to Python-level loop iterations.

5 Discussion

I attempted to build a high-dimensional dataset by adding time as a variable. However, I realized that simply slicing the data by time segments didn't add enough complexity. It essentially started acting like a simple dictionary query rather than a complex spatial search, so it was not included in the final comparison. I also noticed that although the validation checks confirmed that the custom algorithms correctly identified the nearest neighbors, the absolute distance values differed substantially from the GeoPandas baseline. While coordinate projection mismatches were initially suspected, the discrepancy remains unresolved even after verifying that all data was projected correctly.