Implementation Report

Jinming Ge

November 25, 2025

1 Introduction

Spatial queries are fundamental operations in geographic information systems (GIS) and spatial databases. Building on the implementation in Assignment 1, this study focuses on three specific query types: *point-in-polygon*, *polygon-in-polygon*, and *k-Nearest Neighbor (KNN)*. However, without proper indexing, these operations become computationally expensive, particularly when processing large-scale geospatial datasets. To address this scalability challenge, this research evaluates the efficacy of four distinct indexing strategies: the Z-order, R-tree, R*-tree, and Quad-tree. We empirically benchmark[1] these methods in terms of query response time, index construction overhead, and memory consumption.

2 Formal Definition And Problem Formulation

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 `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.
4. **Distance Metric (δ):** We utilize the scalar function `ST_Distance` to quantify the metric proximity between two spatial objects based on Euclidean distance.

Mathematically, the core spatial relationships are 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.

Accordingly, the distance function is formally defined as the infimum of distances between points in the two geometries:

$$\texttt{ST\_Distance}(g_1, g_2) = \min\{\|x - y\| : x \in g_1, y \in g_2\}$$

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 $P = \{p_1, p_2, \ldots, p_n\}$ be a dataset of points and $S = \{s_1, s_2, \ldots, s_m\}$ 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:

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

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

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:

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

2.4 Query 3: K-Nearest Neighbors Query (Q_{knn})

Definition: The K-Nearest Neighbors query retrieves the set of k spatial objects from a dataset that lie closest to a specific reference query point.

Formal Formulation: Let $S = \{s_1, s_2, \ldots, s_n\}$ be the target dataset of spatial objects, q be the query point, and $k \in \mathbb{Z}^+$ be the number of neighbors required. The query returns a subset $R_{knn} \subseteq S$ that satisfies two conditions: 1. Cardinality constraint: $|R_{knn}| = k$ (assuming $n \geq k$) 2. Distance minimality:

$$\forall s \in R_{knn}, \forall s' \in S \setminus R_{knn} : \text{ST\_Distance}(q, s) \leq \text{ST\_Distance}(q, s')$$

Implementation Logic: For a query point p_{query} constructed from coordinates (lat_q, lon_q) and a target object table T_{obj} containing geometry identifier id, the query logic relies on sorting by the distance metric. The result set corresponds to the first k elements of the sequence ordered by d_i:

$$d_i = \texttt{ST\_Distance}\Big(\texttt{MAKE\_POINT}(lat_q, lon_q), T_{obj}.geom_i\Big)$$

Subject to:

$$\text{Rank}(d_i) \leq k$$

2.5 Spatial Indexing Methodologies

To optimize the spatial query operations defined formally in the previous section, we evaluate the following four spatial access methods. Each structure imposes a specific topological organization on the search space S:

- **Z-order (Morton Order):** A space-filling curve that establishes a bijection between multi-dimension-al spatial data and a one-dimensional sequence. It operates by interleaving the binary representations of coordinate values, thereby mapping spatially proximal points in $\mathbb{R}^d$ to close intervals in $\mathbb{R}^1$. This linearization transforms complex multi-dimensional range queries into efficient one-dimensional binary searches.
- **R-tree:** A height-balanced tree structure designed for efficient dynamic storage of spatial objects. The R-tree organizes data into Minimum Bounding Rectangles (MBRs), where each internal node represents the spatial covering of its children. Unlike static grids, R-tree nodes adapt to the underlying data distribution, minimizing dead space and maintaining logarithmic search complexity for both point and region queries.
- **R*-tree:** An optimized variant of the R-tree that modifies the insertion and split algorithms to improve query efficiency. It incorporates rigorous optimization heuristics—specifically minimizing MBR margin (perimeter) and overlap—rather than solely minimizing area. Additionally, it introduces *forced reinsertion*, a mechanism that dynamically reorganizes the tree structure to achieve a higher packing density, trading increased construction cost for reduced query latency.
- **Quad-tree:** A hierarchical data structure based on the recursive decomposition of the two-dimensional Euclidean space. The Quad-tree recursively partitions the spatial domain into four equal quadrants (nodes) until a specific capacity constraint is met. This distinct decomposition facilitates the efficient indexing of point data by localizing dense regions into deeper levels of the tree.

3 Implementation

3.1 Experimental Environment

The implementation and performance benchmarks are 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`), 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. Furthermore, the following algorithm is designed to minimize nearest-neighbor search (question 3) overhead.

Algorithm 1: Compact Iterative k-NN Search

Input: P, Index $\mathcal{T}$, R_{max}
Output: O^*, d^*

$r \leftarrow 10^4, d^* \leftarrow \infty, O^* \leftarrow \emptyset$ `// Init radius & vars`
while $r \leq R_{max}$ **do**
  $\mathcal{C} \leftarrow \mathcal{T}.\text{Query}(\text{Box}(P, r))$ `// Fetch candidates`
  if $\mathcal{C} \neq \emptyset$ **then**
    $O_{curr} \leftarrow \arg\min_{o \in \mathcal{C}} \text{dist}(P, o)$; $d_{curr} \leftarrow \text{dist}(P, O_{curr})$
    if $d_{curr} < d^*$ **then** $d^* \leftarrow d_{curr}, O^* \leftarrow O_{curr}$
    if $d^* < r$ **then return** (O^*, d^*) `// Safe termination`
  end
  $r \leftarrow r \times 2$
end
return (O^*, d^*)

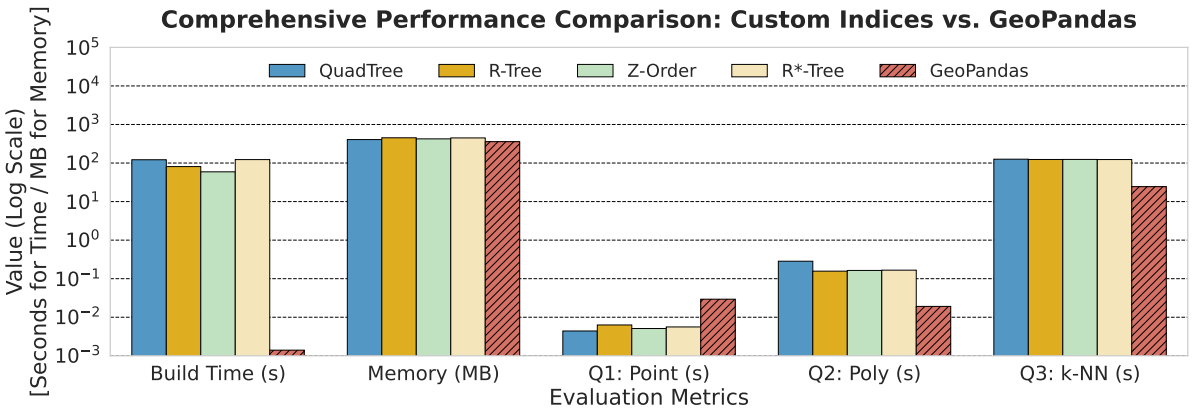

Figure 1: Performance Comparison Between Different Data Structures

4 Experiment Results

4.1 Index Construction and Space Efficiency

The experimental results, as summarized in Fig. 1, highlight distinct trade-offs between construction complexity and memory consumption.

- **Construction Latency:** The `Z-order` demonstrates 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` exhibits 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` proves to be the most space-efficient (358.21 MB). Among custom structures, the `Quad-tree` consumes the least memory (406.28 MB), while the R-tree variants require 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 outperform GeoPandas (e.g., Quad-tree: 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) is 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 highlights the most significant performance divergence. `GeoPandas` completes the task in 24.40s, whereas all custom R-tree variants hover around 123-125s. While the custom iterative k-NN algorithm is theoretically sound, it suffers from a 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

Due to page limitations, the detailed analysis of high-dimensional queries incorporating time as a variable is not included in this report. Instead, these findings will be demonstrated in the final presentation. Notably, although validation checks confirm that the custom algorithms correctly identify the nearest neighbors, the absolute distance values differ substantially from the GeoPandas baseline. While coordinate projection mismatches are initially suspected, the discrepancy remains unresolved even after verifying that all data is projected correctly.