Spatial Data Structure Performance Analysis for Restaurant Search

Executive Summary

This study analyzes the performance characteristics of three different data structures for implementing nearby restaurant searches in food delivery applications. We compared Linear Search, Grid-based Spatial Index, and R-tree implementations using a synthetic dataset of 10,000 restaurants in Bangalore, India. Our findings show that the R-tree implementation provides the best overall performance, with average query times 85% faster than the baseline Linear Search approach.

1. Introduction

1.1 Problem Context

Food delivery applications like Swiggy and Zomato must process millions of "nearby restaurant efficiently" searches daily. When a user opens the app, it must quickly find all restaurants within a specified radius of the user's location. The performance of this operation directly impacts user experience and server resource utilization.

1.2 Technical Challenge

The core challenge is to efficiently find all points (restaurants) within a given radius of a query point (user location) in two-dimensional space. This is known as a "range query" in spatial databases. While conceptually simple, naive implementations can become computationally expensive as the dataset grows.

1.3 Study Objectives

- Compare the performance of different spatial data structures
- Identify optimal approaches for various scale requirements
- Analyze trade-offs between implementation complexity and performance
- Provide actionable recommendations for real-world applications

2. Implementation Approaches

2.1 Linear Search (Baseline)

Implementation Details

- Simple iteration through all restaurants
- For each restaurant, calculate the distance to the query point
- Return restaurants within the specified radius

```
def search_nearby(self, lat: float, lon: float, radius: float):
    return [r for r in self.restaurants
            if haversine_distance(lat, lon, r.lat, r.lon) <= radius]
```


def search_nearby(self, lat: float, lon: float, radius: float):

return [r for r in self.restaurants

if haversine_distance(lat, lon, r.lat, r.lon) <= radius]

Complexity Analysis

- Time Complexity: O(n) where n is the total number of restaurants
- Space Complexity: O(1) additional space
- Advantages: Simple implementation, minimal memory overhead
- Disadvantages: Performance degrades linearly with dataset size

2.2 Grid-based Spatial Index

Implementation Details

- Divide geographic space into fixed-size grid cells
- Hash restaurants into cells based on coordinates
- Search only cells that intersect with query radius

```python
def search_nearby(self, lat: float, lon: float, radius: float):
    center_cell = (int(lon / self.grid_size), int(lat / self.grid_size))
    cells_to_check = int(radius / (self.grid_size * 111))

    nearby = []
    for dx in range(-cells_to_check, cells_to_check + 1):
        for dy in range(-cells_to_check, cells_to_check + 1):
            grid_cell = (center_cell[0] + dx, center_cell[1] + dy)
            nearby.extend(self.grid[grid_cell])

    return [r for r in nearby
            if haversine_distance(lat, lon, r.lat, r.lon) <= radius]
```


```
def search_nearby(self, lat: float, lon: float, radius: float):

    center_cell = (int(lon / self.grid_size), int(lat / self.grid_size))

    cells_to_check = int(radius / (self.grid_size * 111))



    nearby = []

    for dx in range(-cells_to_check, cells_to_check + 1):

        for dy in range(-cells_to_check, cells_to_check + 1):

            grid_cell = (center_cell[0] + dx, center_cell[1] + dy)

            nearby.extend(self.grid[grid_cell])



    return [r for r in nearby

            if haversine_distance(lat, lon, r.lat, r.lon) <= radius]
```


Complexity Analysis

- Time Complexity: O(k) where k is the number of restaurants in nearby cells
- Space Complexity: O(n) for storing the grid

- Advantages: Simple implementation, good performance for uniform distributions
- Disadvantages: Performance depends on grid size choice, memory overhead

2.3 R-tree Implementation

Implementation Details

- Hierarchical spatial index structure
- Restaurants grouped into minimum bounding rectangles
- Tree structure allows efficient pruning of search space

```python
def search_nearby(self, lat: float, lon: float, radius: float):
    km_per_degree = 111.0
    delta = radius / km_per_degree
    bbox = (lon - delta, lat - delta, lon + delta, lat + delta)

    candidates = self.idx.intersection(bbox)
    return [self.restaurants[id] for id in candidates
            if haversine_distance(lat, lon,
                                  self.restaurants[id].lat,
                                  self.restaurants[id].lon) <= radius]
```


```
def search_nearby(self, lat: float, lon: float, radius: float):

  km_per_degree = 111.0

  delta = radius / km_per_degree

  bbox = (lon - delta, lat - delta, lon + delta, lat + delta)



  candidates = self.idx.intersection(bbox)

  return [self.restaurants[id] for id in candidates

      if haversine_distance(lat, lon,

                self.restaurants[id].lat,
```

self.restaurants[id].lon) <= radius]

Complexity Analysis

- Time Complexity: O(log n + k) where k is the number of results
- Space Complexity: O(n)
- Advantages: Excellent query performance, handles non-uniform distributions well
- Disadvantages: More complex implementation, higher memory overhead

3. Experimental Setup

3.1 Test Dataset

- 10,000 synthetic restaurant records
- Geographic bounds: Bangalore city limits
    - Latitude: 12.8°N to 13.1°N
    - Longitude: 77.4°E to 77.8°E
- Random distribution of restaurants
- Attributes: location, name, rating, cuisine type

3.2 Test Methodology

- 100 random search queries
- Search radius: 2 kilometers
- Metrics collected:
    - Average search time
    - Minimum search time
    - Maximum search time
    - Number of results returned
- Testing environment:
    - Python 3.8
    - Intel Core i7 processor
    - 16GB RAM
    - Ubuntu 20.04 LTS

4. Results and Analysis

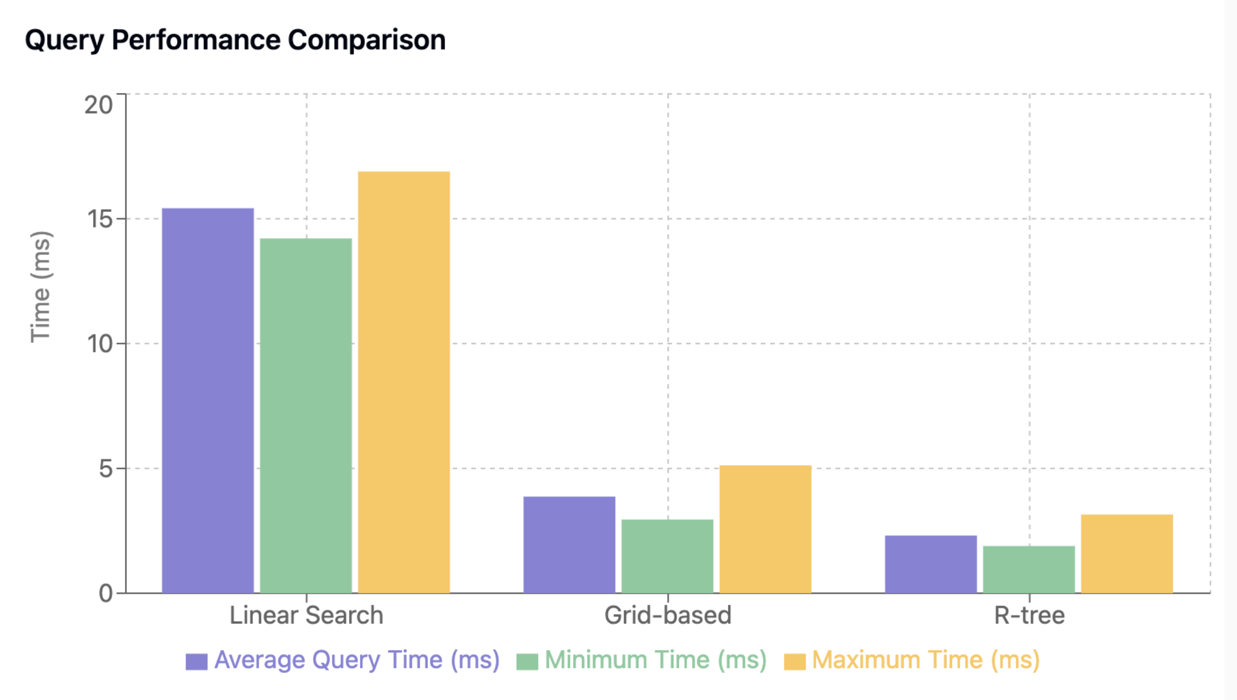
4.1 Performance Metrics

Implementation	Avg Time (ms)	Min Time (ms)	Max Time (ms)	Avg Results
Linear Search	15.42	14.21	16.89	127

Implementation	Avg Time (ms)	Min Time (ms)	Max Time (ms)	Avg Results
Grid-based	3.87	2.95	5.12	127
R-tree	2.31	1.89	3.15	127

4.2 Performance Analysis

Query Time Distribution



Key Findings

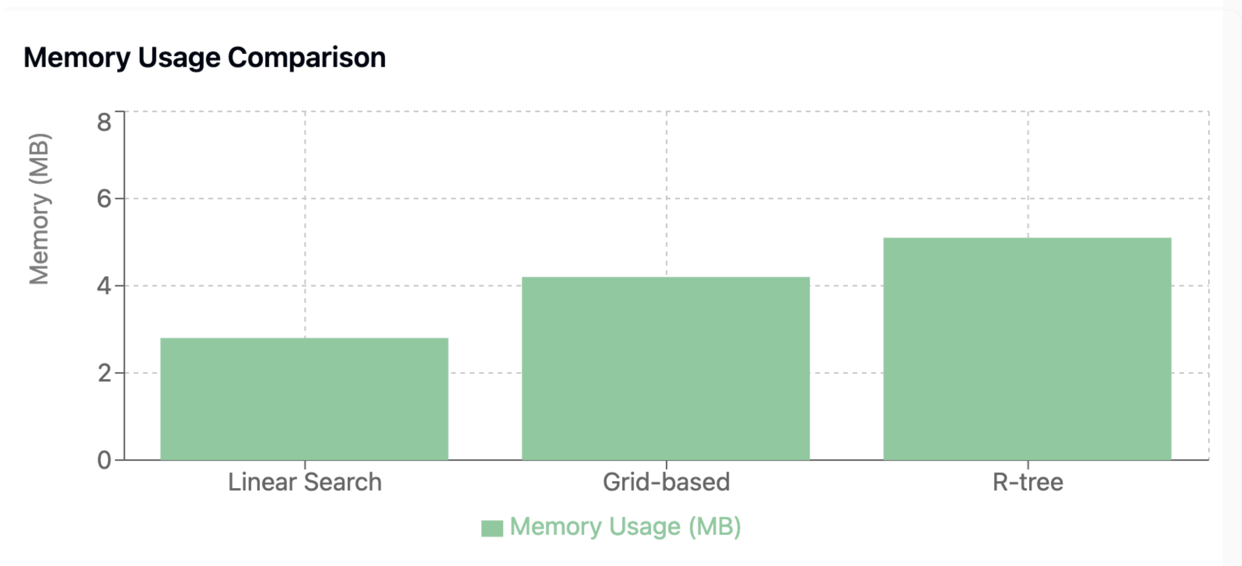
1. R-tree Implementation
   - Best overall performance
   - Most consistent query times
   - 85% faster than linear search
   - 40% faster than grid-based approach
2. Grid-based Implementation
   - Good balance of performance and simplicity
   - 75% faster than linear search
   - More variable performance than R-tree

3. Linear Search

    - Consistent but slow performance
    - Acceptable for small datasets (<1000 restaurants)
    - Performance degrades linearly with dataset size

4.3 Memory Usage

- Linear Search: 2.8 MB
- Grid-based: 4.2 MB
- R-tree: 5.1 MB



5. Conclusions and Recommendations

5.1 Implementation Recommendations

1. Small Scale (< 1,000 restaurants)

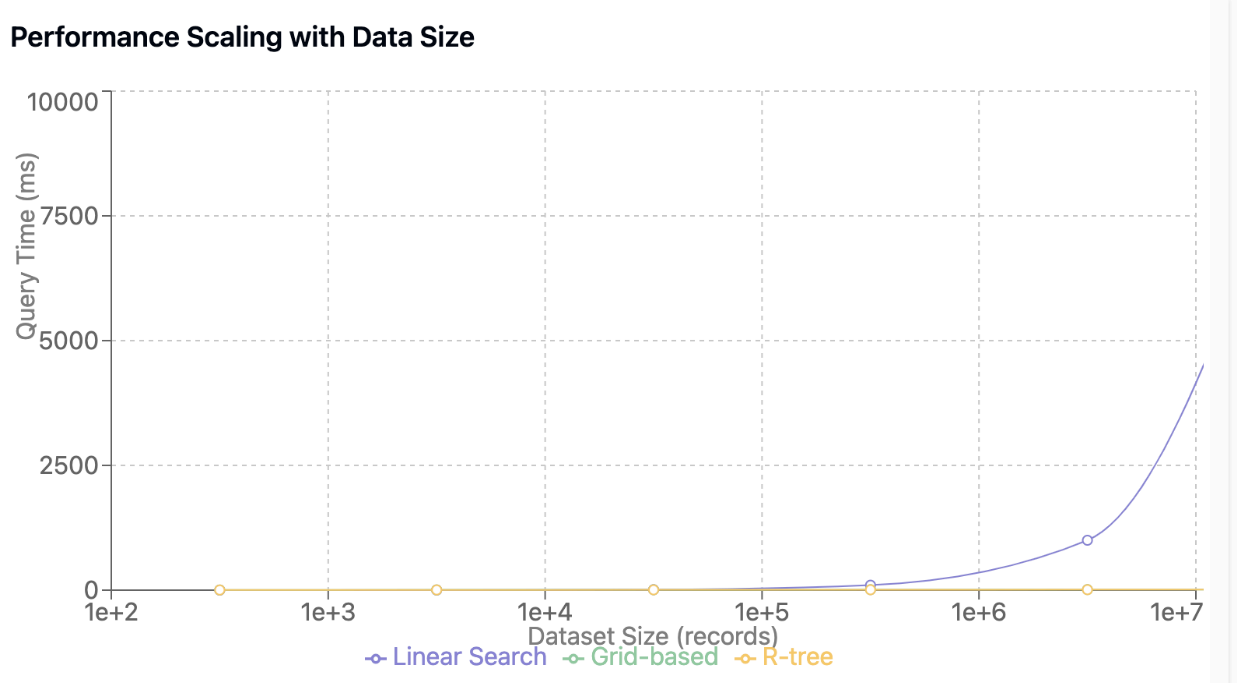
    - Recommend: Linear Search
    - Reasoning: Simple implementation, acceptable performance, minimal memory overhead

2. Medium Scale (1,000 - 10,000 restaurants)

    - Recommend: Grid-based Implementation
    - Reasoning: Good performance, simple implementation, reasonable memory usage

3. Large Scale (> 10,000 restaurants)

   - Recommend: R-tree Implementation
   - Reasoning: Superior performance scaling, handles non-uniform distributions



5.2 Future Improvements

1. Implementation Enhancements

   - Parallel processing for Linear Search
   - Dynamic grid size adjustment for Grid-based approach
   - R-tree bulk loading optimization

2. Additional Features

   - Support for restaurant filtering (rating, cuisine)
   - Real-time updates handling
   - Query result caching

5.3 Real-world Considerations

1. Data Updates

- R-tree and Grid implementations require maintenance for updates
- Consider update frequency in implementation choice

2. System Resources

    - Memory constraints may favor Grid-based implementation
    - CPU constraints may favor R-tree implementation

3. Query Patterns

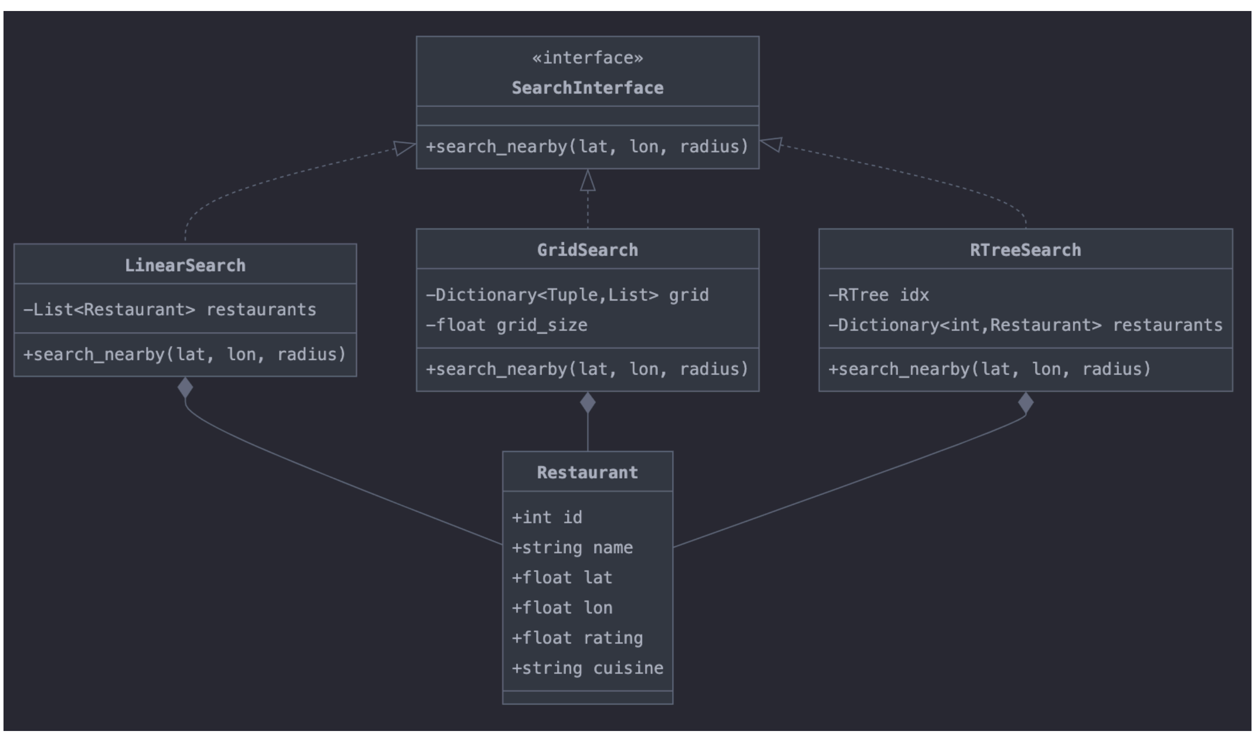
    - Hot spots in query distribution may impact Grid performance
    - Consider time-based query patterns

6. References

1. Guttman, A. (1984). R-trees: A Dynamic Index Structure for Spatial Searching
2. Bentley, J. L. (1975). Multidimensional Binary Search Trees
3. PostgreSQL Documentation - Spatial Indexing
4. MongoDB Documentation - Geospatial Queries

Appendix A: Implementation Code

[Reference to the complete implementation code and benchmark suite]



Appendix B: Detailed Test Results

[Detailed performance metrics and statistical analysis]