


# Optimizing Range Queries with 2D Location Data

Malik Naik Mohammed  
Kennesaw State University  
Marietta, Georgia, USA  
maliknaik16@gmail.com 

**Abstract**—A lot of applications like Uber, Lyft, etc. rely on location data from its users. These applications and services either build their own spatial data management systems or rely on existing solutions [1]. The existing solutions either use a lot of computing power and take a lot of time to execute the code and use inefficient solutions. For this project we implemented the proposed project i.e; performing range queries on location data at scale and accomplished all the milestones along with the algorithm comparison using profiling the code.

**Index Terms**—Geospatial, Range Queries, Google S2, 2D Location, B-Tree, Prefix-tree, and Hilbert-space filling curves.

## I. INTRODUCTION

In recent years, there has been a huge demand for location-based data and companies like Uber, Lyft, Google, etc are investing a lot to improve the access time of location data [1]. The GPS data has attributes like Coordinates (Latitude and Longitude), Altitude, Bearing, and Speed which is not just enough to efficiently query the other GPS integrated devices nearby or within a certain radius. With the increase in the data, fetching the data in an efficient way is necessary, but it's harder to perform the range queries in the 2-Dimensional location data as it might be expensive or inefficient using techniques like Quad-trees or R-Trees. For this project, we'll use the Hilbert curves to map the 2D coordinates onto a 1Dimensional line and use this one dimensional line to perform range queries using B-Trees and Prefix-tree or Radix Tree. The main objective of this project is to perform range queries at scale and within a proximity in an efficient way using B-Tree and Prefix-tree.

## II. LITERATURE REVIEW

In recent years, finding the nearest drivers, businesses or restaurants have fueled an exponential growth in the location-enabled data [1]. Companies like Uber, Lyft, Google, Grab, etc. are investing a lot on either building the spatial management system from scratch or using the existing ones. The main focus of these systems is to scale the search query of large amounts of spatial data. Geospatial queries are very slow and some of these queries require a high amount of compute thereby resulting in scalability issues [3]. The dataset that is commonly used in [1], [2], and [3] is the NYC Taxi Rides dataset that captures all the New York City Rides data until 2015. The paper [1] analyzes various spatial libraries like JTS Topology Suite (JTS), GEOS, Google S2 (S2), ESRI Geometry API, and Java Spatial Index (JSI). Among these libraries Google S2, a spherical geometry library, performs

well when doing the range queries to find the information like finding nearby drivers, restaurants, etc. Spherical geometry is generally well-suited to work with geographic data on a global scale [1]. In [2], Hilbert space-filling curves are used to discretize geographic locations to facilitate the learning of the machine learning model and the querying of the data distributions, so it is efficient to use Hilbert curves to optimize the range queries. In the [1], [2], [3] they are not using the Hilbert space-filling curves to perform range queries and this is a gap in the current literature.

## III. PROBLEM DEFINITION

As discussed in the Introduction, our main objective is to perform range queries at scale and within a proximity with the inputs as raw GPS coordinates. The brute-force (exhaustive) way of achieving this would be to calculate the Euclidean distance between all other devices in the database, but it'll be inefficient as the time complexity is  $O(n)$ , where  $n$  is the number of GPS enabled devices stored in the database. We'll first represent the world/earth on a 2D plane by projection. Our goal is to map the 1-Dimensional line onto the 2D plane and use the 1D line to perform the range queries in the one dimensional line.

The proposed benchmark algorithm uses a huge set of data to process the range queries and each time a query is requested it avoids including the same range as there is a probability of caching the results. The benchmark techniques increase the range query from 10 to 10,000 to the maximum size of the data available. For all these queries, they measure the throughput of each library evaluated in queries/second using the profiling tools. For the distance-based queries, S2 and jvptree performs well and it returns the devices within the  $d$  radius [1].

## IV. SOLUTION

We proposed using the Prefix tree on the CellIds to perform the range-based queries as they perform better than B-trees with the indexing on the CellIds as shown in Fig. 1 and Fig. 2.

How did we achieve this? We'll used the hilbert curve to map the 1 dimensional line onto the 2D plane (see Figure 2)

After successfully mapping this we can store the data and use Prefix-tree to perform the range queries with the CellId. The time complexity of this can be improved to  $O(\log m + k)$ , where  $k$  is the number of nearby locations. We think that this proposed solution works because from the literature survey

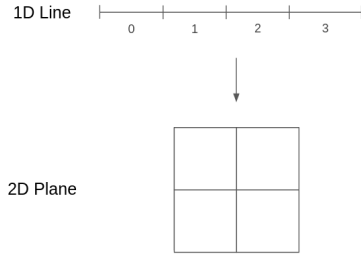


Fig. 1. Mapping the 1D Line on the 2D Plane

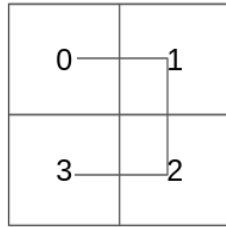


Fig. 2. Mapped 1D line on 2D plane

that we did there is no paper that uses Prefix trees to perform the range-based queries.

The time and space complexity of B-Tree for this problem is as follows: Time Complexity:  $O(k \log n)$  Space Complexity:  $O(n)$  The time and space complexity of Prefix-Tree/Trie/Radix-tree for this problem is as follows: Time Complexity:  $O(\log m + k)$  Space Complexity:  $O(n)$

Where,  $n$  = Number of records  $m$  = Length of the CellId  $k$  = Number of nearby locations

## V. IMPLEMENTATION

We implemented this project in Python and used SQLite on a Linux machine. In order to implement the hibert mapping we use the Google S2 library, spherical geometry library, to map the 1D line onto the 2D plane. We used the Fast Food Restaurants in America Dataset [4] that contain the location data (with lat/long) of all the restaurants in the United States and perform range queries. We even profiled the code using a profiling library in python called cProfile. The cProfile does some statistical analysis of the python code and returns useful information like how often and for how long various parts of the program are executed [6]. We wrote our performance comparison using Matplotlib for plotting graphs.

The following is a part of the code that was implemented to perform spatial queries using Google S2 (truncated for better readability):

```
1 import pywraps2 as s2
2
3 class Spatial:
4     def __init__(self, min_level=10, max_level=10,
5         max_cells=50, is_btree=True):
6         """
7         Initialize the Spatial object.
8         """
9         # ...
10
11     def get_coverer(self):
12         # ...
13
14
15
16
17     def near(self, point, radius, return_tokens=
18         False, units='mi'):
19         """
20         Returns the nearby CellIds from the given
21         point around the given radius.
22         """
23
24         # Get the Coverer.
25         coverer = self.get_coverer()
26         # ...
27         # Get the angle in radians.
28         angle = s2.S1Angle.Radians(float(radius) /
29             6371000)
30
31         # Convert the Latitude and Longitude to
32         S2Point.
33         point = s2.S2LatLng.FromDegrees(point[0],
34             point[1]).ToPoint()
35
36         # Get the S2Cap from the given point.
37         sphere_cap = s2.S2Cap(point, angle)
38
39         # Get the covering around the region.
40         covering = coverer.GetCovering(sphere_cap)
41
42         if return_tokens:
43             return list(map(lambda x: x.ToToken(),
44                 covering))
45
46         return covering
47
48     # ...
49     def get_bounding_rect(self, token):
50         """
51         Returns the bounding latitudes and longitudes
52         for the given cellId.
53         """
54
55         bounds = []
56         token = s2.S2CellId.FromToken(token)
57         rect = s2.S2Polygon(s2.S2Cell(token)).
58             GetRectBound()
59
60         for i in range(4):
61             vertex = []
62             vertex.append(rect.GetVertex(i).lat().
63                 degrees())
64             vertex.append(rect.GetVertex(i).lng().
65                 degrees())
66             bounds.append(vertex)
67
68         return bounds
```

Listing 1. Python example

We have pushed the code to github and can be found here: <https://github.com/maliknaik16/range-queries-location>

The Fig. 3 and Fig. 4 shows the homepage of the developed

## User Coordinates

Lat:

Long:

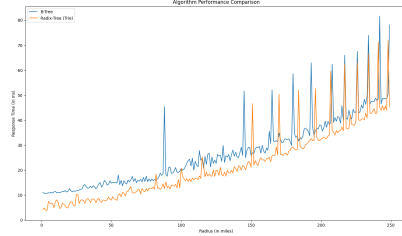
Radius:

Default coordinates are of [Kennesaw State University - Marietta Campus](#) with radius of 1 mile

[View Nearby Restaurants](#)

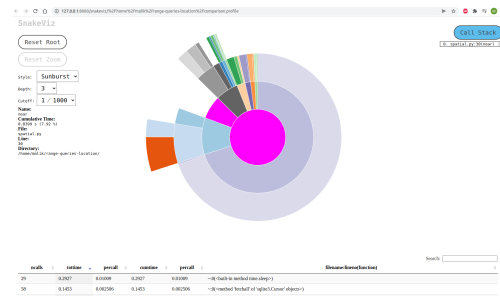
## VI. PERFORMANCE EVALUATION

From Fig. 6 we can find the previous response time of the query using B-tree and Prefix tree indexing with 10,000 restaurant records. We observe that the Radix/Prefix tree returns responses slightly better than the B-tree indexing. The spikes in the graph for some queries is due to running other applications on the laptop. The simulation was done on Intel(R) Core(TM) i7 CPU @ 2.80GHz (4 Cores) with 8GB RAM on Linux Mint 20 Ulyana (Debian-based).



## VII. PROGNOSIS

We analyzed the implemented code using cProfile and visualized the function call time with SnakeViz in SunBurst style shown in Fig. 7. Most of the calls were made on the `get_nearby_locations()` method in the Spatial class.



We've learnt a lot about analyzing the algorithms and how to use the right algorithm to the right problem to make the best use of it. Also, we learnt to analyze the algorithms and compare them with the other algorithms and see how it performs with increasing data or load.

For this project, we implemented and compared the proposed algorithms. Visualized the prefix trees using NetworkX. We even made the UI (User Interface) to display/view the nearby restaurants on the Maps using Leaflet and Web App (Flask). This work can be further extended by performing the range queries using K nearest neighbors (kNN), Quad-trees, etc.

## REFERENCES

- [1] Pandey, V., van Renen, A., Kipf, A. et al. How Good Are Modern Spatial Libraries?. *Data Sci. Eng.* 6, 192–208 (2021). <https://doi.org/10.1007/s41019-020-00147-9>
- [2] Dimitri Vorona, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2019. DeepSPACE: Approximate Geospatial Query Processing with Deep Learning. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19)*. Association for Computing Machinery, New York, NY, USA, 500–503. DOI:<https://doi.org/10.1145/3347146.3359112>
- [3] Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. 2016. High-Performance Geospatial Analytics in HyPerSpace. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 2145–2148. DOI:<https://doi.org/10.1145/2882903.2899412>
- [4] Rishi Damarla. 2021. Fast Food Restaurants in America. (July 2021). Retrieved December 2, 2021 from <https://www.kaggle.com/rishidamarla/fast-food-restaurants-in-america>.
- [5] Christian S. Perone. 2015. Google's S2, geometry on the sphere, cells and Hilbert curve. (August 2015). Retrieved September 2, 2021 from <https://blog.christianperone.com/2015/08/googles-s2-geometry-on-the-sphere-cells-and-hilbert-curve/>.
- [6] Python. 2021. The Python Profilers. (September 2021). Retrieved September 9, 2021 from <https://docs.python.org/3/library/profile.html>