Erik Mjaaland Skår

Exploring Linked List-based Trajectory Traversal for Efficient Topological Queries in Spatial Data

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2023

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Erik Mjaaland Skår

Exploring Linked List-based Trajectory Traversal for Efficient Topological Queries in Spatial Data

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Norwegian University of
Science and Technology

Abstract

The rapid growth of spatial data due to the proliferation of IoT devices and smartphones presents new challenges for spatial data management. Spatial databases have been effective in querying spatial data for several decades, but topological queries are often considered computationally expensive, and existing data structures that improve such queries typically sacrifice performance on spatial queries. In this thesis, we aim to enhance the efficiency of topological queries in spatial data without altering the underlying data structure. To achieve this, we propose using linked lists to traverse trajectories as a graph, thereby enabling efficient identification of trajectory heads and tails without requiring expensive lookups. We conducted benchmarking tests that show the proposed method to be significantly faster than previous approaches, indicating that it merits further exploration. This study contributes to the field of spatial data management by presenting a novel approach to improve the efficiency of topological queries, which could have practical implications for real-world applications.

Sammendrag

Den raske veksten av romlig data grunnet økt tilgjengelighet av IoT-enheter og smarttelefoner presenterer nye utfordringer for romlig datahåndtering. Databasesystemer designet for romlig data har vært effektive i spørring av romlig data i flere tiår, men imidlertid anses topologiske spørringer ofte som beregningsmessig dyre, og eksisterende datastrukturer som forbedrer slike spørringer, ofrer ofte ytelse på romlige spørringer. I denne oppgaven har vi som mål å forbedre effektiviteten av topologiske spørringer av romlig data uten å endre den underliggende datastrukturen. For å oppnå dette, foreslår vi å bruke lenkede lister til å traversere spor av noder som en graf, og dermed muliggjøre effektiv identifikasjon av start- og sluttnoder uten å måtte ha en indeks eller full tabellskanning. Vi gjennomførte ytelsestester som viser at den foreslåtte metoden er betydelig raskere enn tidligere tilnærminger, noe som indikerer at den fortjener ytterligere utforskning. Denne studien bidrar til feltet for romlig datahåndtering ved å presentere en ny tilnærming for å forbedre effektiviteten av topologiske spørringer, som kan ha praktiske implikasjoner for virkelige applikasjoner.

Preface

This thesis is written as a part of the subject *IT3920 - Master Thesis for MSIT* under the supervision of Svein Erik Bratsberg.

Acknowledgements

I would like to thank my friends and family for supporting me throughout the process of writing this thesis. Additionally, I would like to thank my supervisor Svein Erik Bratsberg for guiding me on this journey and providing valuable feedback.

Disclaimer

This thesis builds upon the work done in the subject *IT3915 - Master in Informatics, Preliminary Project*. Because of this, some sections contain similar content. Specifically Section 2.2 will contain some content from the previous report.

Contents

1 Introduction **1**
- 1.1 Chapter overview . . . 3
- 1.2 Abbreviations . . . 3

2 Background **5**
- 2.1 Indexes . . . 5
- 2.2 Spatial Data . . . 6
  - 2.2.1 Existing Data Structures for Spatial Data . . . 6
  - 2.2.2 Quadtrees . . . 8
  - 2.2.3 R-trees . . . 9
  - 2.2.4 Space-filling Curves . . . 13
- 2.3 Linked Lists . . . 15
- 2.4 Querying Spatial Data . . . 16
- 2.5 Combining R-trees with linked lists . . . 17
  - 2.5.1 Advantages . . . 18
  - 2.5.2 Disadvantages . . . 19
- 2.6 Related Work . . . 19

3 Method **21**
- 3.1 Benchmarking Performance . . . 23
- 3.2 R-tree Parameters . . . 24
  - 3.2.1 Special considerations . . . 25
- 3.3 Choice of Dataset . . . 27
- 3.4 Benchmarking . . . 29
  - 3.4.1 Number of data points . . . 30
- 3.5 Queries . . . 30
  - 3.5.1 Start and End Query . . . 31
  - 3.5.2 Crosses Query . . . 32

3.5.3 Enters and Leaves Query . . . 33
3.6 Environment . . . 34
3.7 Preventing Anomalies . . . 34
3.8 Reducing Search Space . . . 35

4 Results **37**
4.1 Reproducability . . . 40

5 Discussion **41**

6 Conclusion **45**
6.1 Future Work . . . 45
6.2 Conclusion . . . 46

A Additional Figures **51**

B Scripts **55**

C Data **57**

List of Figures

1.1 A trajectory as presented in the Strava application on iOS. 2

2.1 Map Showing Pace Using Spatiotemporal Data 7
2.2 PR Quadtree 9
2.3 TB-tree 11
2.4 R-tree 12
2.5 Hilbert Space-Filling Curve 14

3.1 Visualization of different approaches 22
3.2 Number of Nodes and Leaf Nodes 24
3.3 Area vs Circumference MBR 27
3.4 Streets of Porto Visualized 29
3.5 Index Visualized 30
3.6 Queries Visualized 31
3.7 Start and End Regions Visualized 33

4.1 Median Query Time 38
4.2 Linked List Median Query Time 38
4.3 Index Median Query Time 39
4.4 Iterative Search Median Query Time 39

5.1 Range Search Max and Median Query Time 43

A.1 Median Start and End Query Time 52
A.2 Median Crosses Query Time 52
A.3 Median Enters Query Time 53
A.4 Median Leaves Query Time 53

List of Tables

1.1 Abbreviations 4

2.1 Comparison of Various R-tree Variations 11
2.2 Strengths and Weaknesses of Spatial Partitioning Techniques 14
2.3 Types of Spatiotemporal Queries 16

3.1 Summary of R-tree Parameters 26
3.2 Considered Datasets 29

C.1 Median All Queries 58
C.2 R-tree Implementation Data 61
C.3 Range Search Response Time 62

Chapter 1

Introduction

The integration of the Internet of Things (IoT) into modern society has become increasingly widespread in recent years, leading to a surge in the amount of data generated daily, particularly spatial data. McKinsey estimates that IoT has an economic impact of \$3.9 trillion \$11.1 trillion per year in 2025 [1, p. 2]. Notably, companies like Strava have become household names, earning millions of dollars by generating, storing, and aggregating spatial data for their users [2]. Typically, this type of data is presented as trajectories in the form of linked directional points in two or three dimensions that depict the path of the user. An example of this can be seen in Figure 1.1. While spatial databases have traditionally been efficient at querying data based on their spatial properties, performing *topological* queries, which extract information by examining the trajectory as a whole, is still considered to be computationally expensive [3]. Topological queries are closer defined in Section 2.4. To address this issue, data structures such as *TB-trees* have been developed as R-tree variants that prioritize preserving trajectory segments when splitting rather than the spatial properties of the data points. This results in a trade-off that negatively impacts the ability to effectively query spatial properties and topological properties simultaneously without the need for several indexes.

This paper proposes utilizing linked lists to enhance topological queries without modifying the underlying indexing data structure, but by modifying the data itself. We aim to solve the research question ***"can linked lists be utilized to improve topological query performance?"***. To facilitate this we need to complete the following objectives:

1. Find a suitable dataset for experimentation. This will be closer described in Section 3.3.

2. Implement an underlying indexing method to allow for range searches.

3. Compare the response time for topological queries performed using iterative, indexed

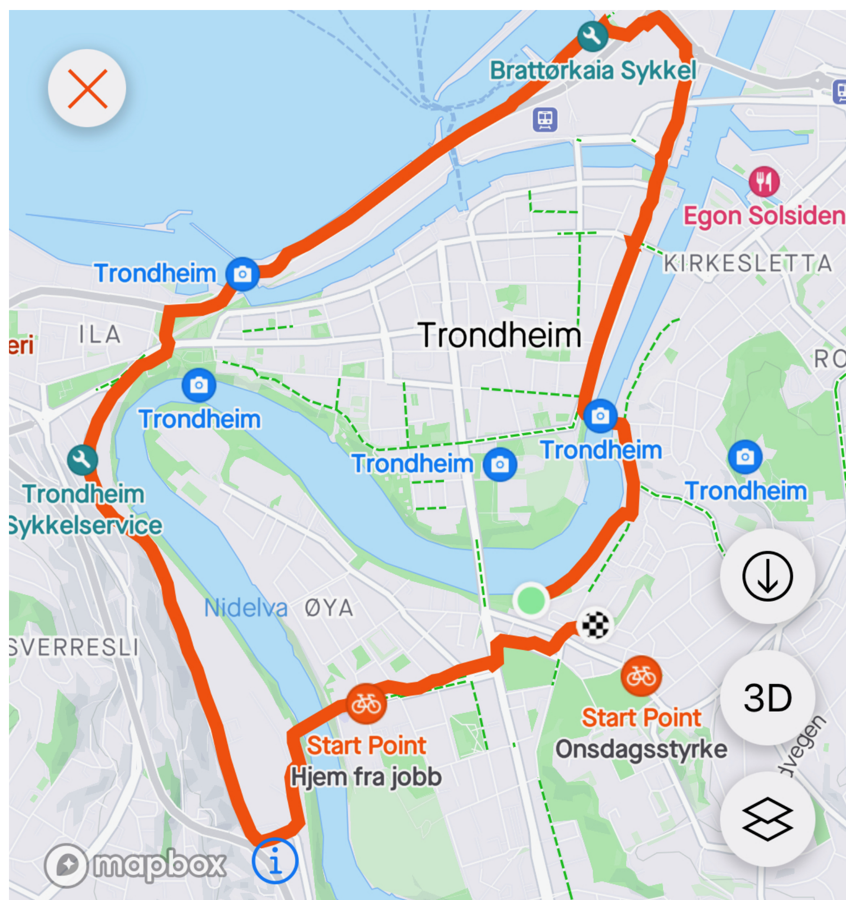


Figure 1.1: A trajectory as presented in the Strava application on iOS.

and linked list approaches to determine which has the best performance.

1.1 Chapter overview

This thesis is structured into several chapters, with the second chapter providing background information about spatial data, existing data structures and indexing methods, which will be presented, discussed, and compared. Furthermore, the context will be further explained, and related work will also be discussed in this chapter. The third chapter will focus on the methodology employed in this project, where we will introduce our new implementation and describe how the benchmarking process was conducted, including our choice of technology and dataset. In Chapter 4, we will present the results of our benchmarking and compare them to the predicted outcomes. These will be discussed further in Chapter 5. Finally, we will conclude, discuss further work and the implications of our findings in the last chapter.

1.2 Abbreviations

Throughout this thesis, there will be some abbreviations used. Table 1.1 shows all the abbreviations and their meaning.

Table 1.1: Abbreviations

Abbreviation	**Meaning**
API	Application Programming Interface
C	Crosses query
DBMS	Database Management System
DE-9IM	The Dimensionally Extended 9-Intersection Model
E	Enters query
GIS	Geographic Information Systems
GPS	Global Positioning System
ID	Identification
IN	Indexed (query variation)
IoT	Internet of Things
IT	Iterative (query variation)
JVM	Java Virtual Machine
k	Number of datapoints within a region
l	Leaf node
L	Leaves query
LL	Linked List (query variation)
maxFill	Maximum Filling Factor
MBR	Minimum Bounding Rectangle
minFill	Minimum Filling Grade
N	Total number of data points
n	Node
OGC	The Open Geospatial Consortium
OLAP	Online Analytical Processing
RDBMS	Relational Database Management System
SE	Start and End query
SFC	Space-Filling Curve
T	Trajectory
TB-Tree	Trajectory Bundling Tree
W	Query Window

Chapter 2

Background

Some context and knowledge are required for understanding what this thesis aims to solve. It is assumed that the reader has some knowledge of the topic of database management systems and spatial data, but a brief introduction will be given in this chapter. For a more detailed explanation of spatial data and its underlying data structures, materials such as *The Design and Analysis of Spatial Data Structures* are recommended [4; 5; 6; 7; 8; 9; 10].

2.1 Indexes

Database indexing is a fundamental concept in database management systems (DBMS) that involves organizing data in a manner that facilitates efficient retrieval of information [9]. The efficiency of data retrieval is critical for the performance of database applications, particularly those that handle large amounts of data. The type of index depends on the type of data, as well as what fields of the data are frequently accessed. DBMSs such as *MySQL* use B-trees as the default when creating an index, with some exceptions such as using R-trees when indexing spatial data [11]. For search engines using full-text search, it would be beneficial to use another index known as a reverse index as it is better suited for text retrieval.

Indexes can be either clustered or nonclustered. Clustered indexes, often referred to as primary indexes, are indexes where the data is stored directly in the index as opposed to nonclustered or secondary indexes where the data is stored separately, but the index contains pointers to the data [9, p. 631]. As the data is stored directly in the indexing structure when using clustered indexes it can only have one clustered index, but it can also have nonclustered indexes pointing to the data in the clustered index. With nonclustered indexes, one can have several indexes for the data as they are simply pointers to the data. For instance, if you have data with spatial properties, as well as other properties,

you can have one index such as a B-tree for the non-spatial properties and an R-tree for the spatial properties. Spatial indexes are most commonly nonclustered as there are non-spatial fields associated with the data, which are often preferred to have physically close together to improve join query performance.

2.2 Spatial Data

Spatial data is simply data that contains spatial properties in some shape or form. These can include but are not limited to, points, lines, rectangles, regions, surfaces and volumes [4]. There can be additional data present, but that is not a requirement. One of the most common data types to pair with spatial data is temporal data. This is known as spatiotemporal data and is used extensively throughout different fields such as geographic information science, climate science, public health and logistics [12; 13; 14; 15]. One interesting effect of storing spatial and temporal data together is that you can create trajectories, essentially drawing a line from the starting point, through all the intermittent points and to the endpoint. To know when to save the user's location the two most common methods are to either save the location at a set time interval or to save the location after a set distance interval. Another way, commonly used by apps such as Google Maps, is to save the location when the user moves a lot or uses apps that use GPS signals [16]. From the GPS positions alongside a timestamp, you can see how long each stop is, the average movement speed, where the user spends most of their time and generate patterns in user movements. This data can be very valuable to companies that can use this data to precisely target relevant users. In Figure 2.1 the same trajectory as shown in Figure 1.1 has overlaid colors showing the pace at different parts of the trajectory.

To limit the scope of this project, we will not focus on the temporal part of the data. We will instead only look at the spatial properties, such as the coordinates, but we can use the temporal properties to induce an ordering of the points, creating a linked directional property to each trajectory. We will also not focus on spatial data with more than two dimensions or look into implementations in specific databases.

2.2.1 Existing Data Structures for Spatial Data

Spatial data has been around for many decades. As early as the 1960's research began on studying geographic information systems (GIS) [17]. However, as the amount of spatial data has increased significantly over the past few years, due to the rise of smartphones and the Internet of Things (IoT), spatial data has become more relevant than ever [18].

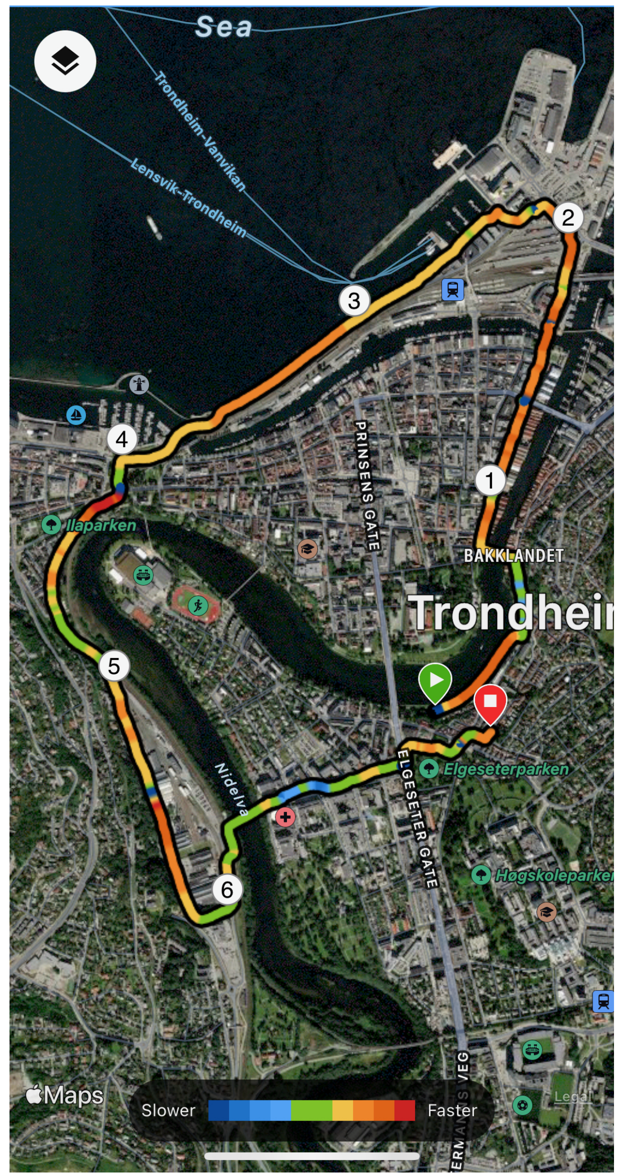


Figure 2.1: A screenshot from the Garmin Connect app on iOS of the same trajectory as in Figure 1.1. The different colors show the speed at the given part of the trajectory.

The most common way to store and work with spatial data is through spatial databases. These are databases that are specifically designed to store and perform queries on spatial data. For a spatial database to store and query spatial data, some properties need to be present. The most important one is the spatial indexing method, as that is essential for querying spatial data effectively. Normal databases use indexes such as B^+-trees, which provide efficient queries on ordered data, but these are not efficient for querying spatial data [9, p. 652]. Most spatial database systems allow for using different indexing methods based on what the user needs, but the most common indexing method is R-trees or any of its variants. The different indexing methods all take different approaches and prioritize differently to accommodate different use cases. Some prioritize dynamic data, data that will change during usage, and some prioritize static data. When prioritizing dynamic data, an index method must easily be recreated or updated to accommodate the changes. If it takes a long time to adapt to the changes, the queries to the data will be outdated or the service will experience much downtime. If one knows that there are not going to be any changes to the data, one can go for an indexing method that might use more time to generate the index but has a faster time for querying the data after the index has been generated.

There are two main categories for the data structures used by spatial indexing methods: space-driven and data-driven structures. Space-driven structures divide the entire space into zones based on the data present. Some examples of this include quadtrees and kd-trees. Data-driven structures, on the other hand, do not divide the entire space, but rather use the data to only create zones that are encompassing the data, using only the minimum amount of space required. This category contains R-trees and all their variants.

Different indexing methods can also be faster for some types of queries and slower for others. The *Open Geospatial Consortium* has defined some predicates which indicate how geometries interact with each other [19]. These and others are often built into the spatial databases such that database managers do not have to write custom functionalities for them.

2.2.2 Quadtrees

Quadtrees are a space-driven data structure used for indexing two-dimensional spatial data. They work by dividing the space into four squares, and then dividing those again until you only encompass a set amount of points in each square. There are several variations of quadtrees as they are used for different use cases such as image compression, storing polygons and several others. The different variations include but are not limited to, point-, region, point-region-, PMR- and XBR quadtrees [20]. In Figure 2.2 you can

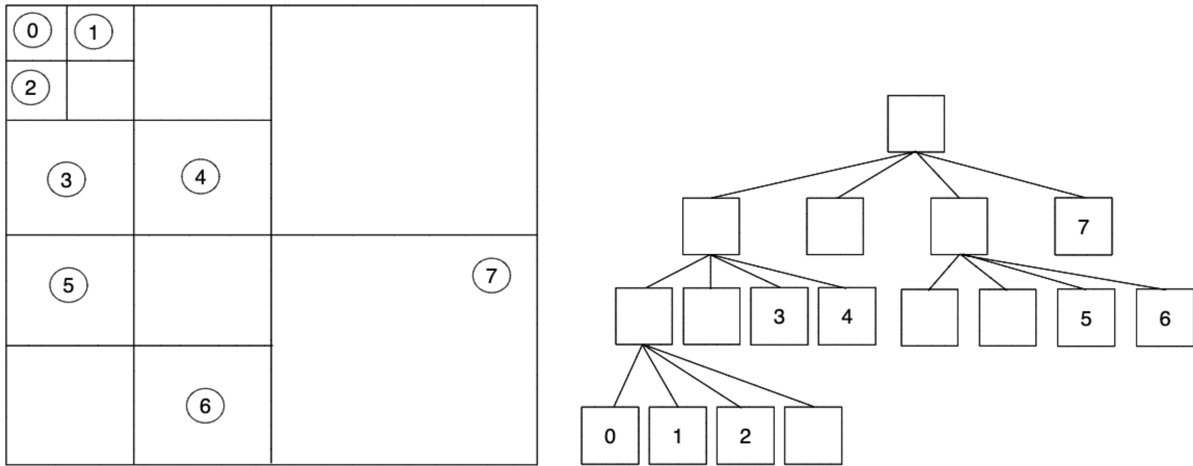


Figure 2.2: A PR quadtree with seven points and a maximum of one point per cell represented as both a grid and a tree.

see a point-region (PR) quadtree represented as both a grid based on the position of the points as well as a tree. For connected directional spatial data, the most relevant quadtrees to look at would be the PMR- and XBR quadtrees as they are specifically designed to store and query trajectory data [21].

Variants of this data structure are used by companies such as Strava, where they use quadtrees to aggregate data such as a heatmap of each user's activities [22]. Their quadtrees have several layers of granularity which update as the user zooms on the map. This is quite similar to the functionality of the *drill-down* operation in an OLAP cube. In their blog post, they do not specify why they went with a space-driven data structure instead of a data-driven data structure, but it is probably a culmination of the factors Oracle found in their testing which will be discussed shortly.

2.2.3 R-trees

R-tree is a data-driven structure invented in 1984 by Antonin Guttman that is used for indexing multi-dimensional spatial data [6]. The concept is closely related to B$^+$-trees, but the data is grouped by its spatial properties. It works by grouping items by using *minimum bounding rectangles* (MBR). MBRs are the smallest possible rectangles one can create for a set of objects whilst enveloping their spatial properties. These are created or updated whenever data is inserted into the tree. Each leaf node can contain a set amount of data, usually denoted as its *maxFill* property, and require a minimum amount of data known as its *minFill* property. Non-leaf nodes have a property named *fanout* which determines how many sub-regions it can contain before overflowing. Whenever

a leaf node is overflowing, the node will be split. Different variations often use different splitting algorithms, but the most common ones in the default R-tree are linear, quadratic and exponential splitting [10]. Each has a distinct prioritization of either speed or better splits, with linear being the fastest with the worst splits and exponential being the slowest with optimal splitting. The choice of splitting algorithm is often dependent on if the tree is to be often updated (dynamic) or if it is only to be built once (static). Whenever a leaf node is split, the split is propagated upwards and each parent node will be split if the number of sub-regions exceeds the limit set by the fanout. If the root exceeds the fanout it will be split, setting a new root node as its parent and the tree has increased its depth by one. Because the splits result in two new nodes the tree is self-balancing which is beneficial in maintaining its query efficiency.

Throughout time there have been many proposed variations of R-trees, with a few of them being used commercially. R*-trees were first proposed in 1990 and improved on the original concept [23]. In essence, the R*-tree uses more computational power to create trees with less overlap and smaller MBRs. This allows for better query performance than the original R-tree. The R^+-tree does not allow overlapping MBRs on the same level, causing greater performance for point queries, but worse performance for window queries due to storing duplicates rather than overlapping MBRs. In addition to these variants, several others do their optimizations and concessions to improve the performance in some regard. A more detailed and comprehensive list of R-tree variations can be found in [7].

There exist other R-trees designed especially for trajectories. One of these includes the *Trajectory Bundling Tree*, also known as a TB-tree, which was first proposed alongside the STR-tree in the article "Novel Approaches to the Indexing of Moving Object Trajectories" released in 2001 [3]. The TB-tree seeks to improve the performance of queries that retrieve trajectories as it stores segments of each trajectory bundled together in each leaf node. This sacrifices some query performance for the normal range and point searches, as MBRs of trajectory segments will not necessarily be as small as possible, in favor of making trajectory queries faster through the locality of the data. In addition to this, each leaf node containing a trajectory segment will have a pointer to the previous segment as well as the next segment. This creates a doubly linked list of trajectory segments, making retrieval of entire segments trivial. These links are not without their caveats, one of them being that whenever a node is split, all pointers pointing to it would need to be updated. Figure 2.3 visualizes the structure of the TB-tree and Table 2.1 shows the strength and weaknesses of the different R-tree variations.

A study done by Oracle in 2002 found that R-trees are faster than quadtrees for almost

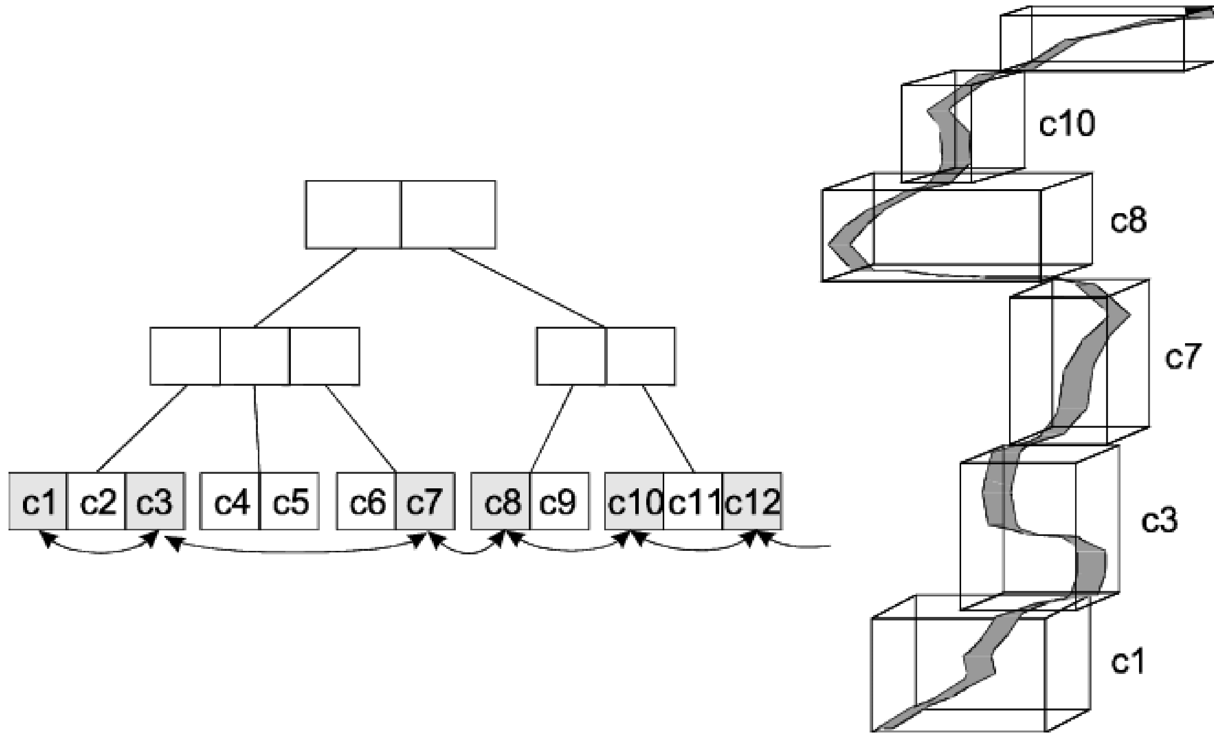
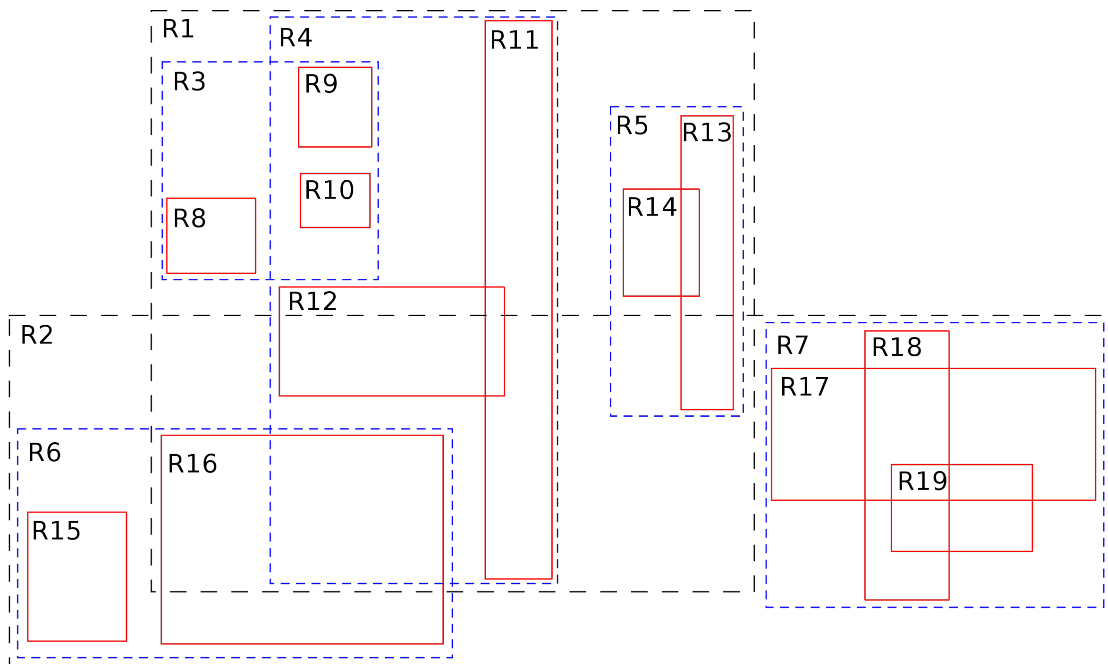


Figure 2.3: The TB-tree structure [3].

Table 2.1: Comparison of various R-tree variations.

Name	**Dynamic / Static**	**Index Size**	**Overlap**	**Disk Usage**	**Query Efficiency**	**Topological Queries**
R-tree	Dynamic	Medium	High	Medium	Good (Range, NN)	Fair
R*-tree	Dynamic	Medium	Low	Medium	Very Good (Range, NN)	Fair
R+-tree	Dynamic	Low	None	Low	Fair (Range, NN)	Fair
Hilbert R-tree	Dynamic	Medium	Low	Medium	Very Good (Range, NN)	Fair
TB-tree	Dynamic	High	Low	High	Excellent (Trajectory)	Excellent



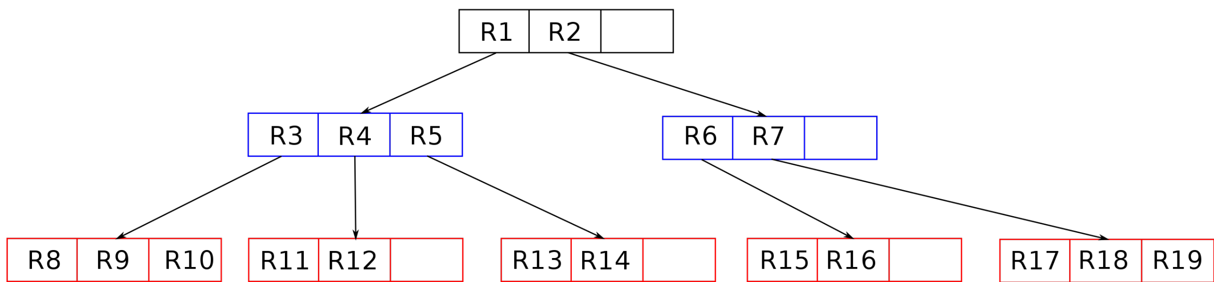


Figure 2.4: An example of an R-tree for two-dimensional rectangles by Stefan de Konink [24].

all spatial queries. R-trees were faster for *anyinteract, inside, contains, touch, coveredby, covers, equals, overlapbydisjoint*, and *overlapbyintersect* queries. However, quadtrees were better for *touch, overlapby, disjoint,* and *overlapbyintersect* queries only when using larger query windows. They also found update times to be better for R-trees and storage requirements similar for both structures, but only when using points. For everything else R-trees outperform quadtrees. Essentially, they recommend using R-trees unless you are creating "[...] update-intensive applications using simple polygon geometries, high concurrency update databases, or when specialized masks such as touch are frequently used in queries" [25, p. 555].

They also found that using quadtrees required a lot of fine-tuning to reach optimal performance. This is not required for R-trees and provides R-trees with another advantage.

Quadtrees are usually preferred for evenly distributed data where updates are frequent due to the nature of how changes do not need to propagate as far up the data structure as with R-trees.

2.2.4 Space-filling Curves

Space-filling curves (SFCs) are mathematical functions used to map multi-dimensional data points to a single dimension whilst preserving the spatial relationships of the data points. They only access each location once, and never cross themselves [26]. SFCs have several applications including image compression, computer graphics and spatial indexing. There are several different variations of SFCs, with the most well-known being the Hilbert curve which was described in 1891 by David Hilbert [27]. It is preferred over other curves, such as the Morton curve, as it better preserves the spatial relationships between points [4].

Space-filling curves can be utilized in spatial indexing either alone or accompanied by either space-driven or data-driven structures such as the ones introduced previously. The Hilbert R-tree is a known variation of the R-tree that utilizes the Hilbert space-filling curve to better preserve the locality of the data points than the original R-tree in case of splits, making I/O faster. Space-filling curves can also be used in conjunction with quadtrees as can be seen in Figure 2.5. Table 2.2 shows the strengths and weaknesses of the different spatial partitioning methods.

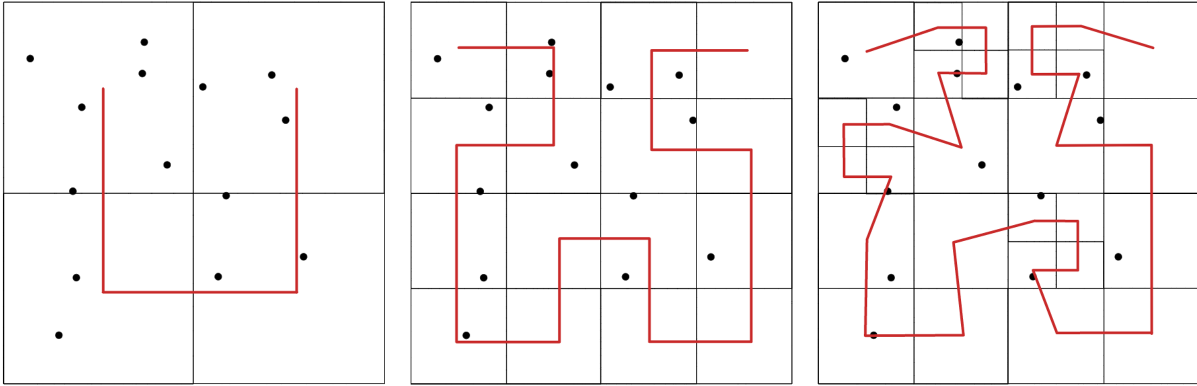


Figure 2.5: Hilbert space-filling curve used with a quadtree. Note how the depth of the quadtree affects the order of the curve.

Table 2.2: Some of the strengths and weaknesses of different spatial partitioning techniques.

Partitioning Type	**Strengths**	**Weaknesses**
Space-Driven Partitioning (e.g., Quadtrees)	1. Simple and easy to implement. 2. Very efficient for dynamic data. 3. Many levels of granularity.	1. Not suitable for highly clustered data. 2. Requires tweaking to be efficient. 3. Not self-balancing.
Data-Driven Partitioning (e.g., R-Trees)	1. Efficient for multi-dimensional range and nearest neighbor queries. 2. Self-balancing (most of them). 3. Highly adaptable.	1. High complexity of implementing and maintaining the data structures. 2. Deletions can be inefficient. 3. May require rebalancing or rebuilding for efficient queries.
Space-Filling Curves (e.g., Z-Order, Hilbert Curves)	1. Efficiently maps multi-dimensional data to one dimension. 2. Preserves the locality of the data. 3. Simplifies handling of multi-dimensional data.	1. Some queries can be complex to execute. 2. Handling non-uniform data can be inefficient. 3. Mapping can lead to the loss of multi-dimensional characteristics.

2.3 Linked Lists

Linked lists are a fundamental data structure in computer science [28, p. 25]. It consists of nodes linked together with pointers pointing from one node to the next. Nodes cannot be accessed through indexes as the nodes can be stored anywhere in memory, with only the pointers from the previous node telling where the data is stored in memory. Because of this you rather have to traverse the list from start to finish to read items. Linked directional data is, depending on the implementation, essentially just linked lists consisting of nodes with spatial properties. Some nodes are logically connected to other nodes through some kind of order. This ordering can be either explicit, by having them stored with references to each other, or implicit by having them either stored in a specific order or by fields such as timestamps. Utilizing linked lists for spatial data is not a novel thought and has been used in MX-CIF quadtrees previously in 1982 as well as in TB-trees in 2001 [4; 3].

Linked lists can either be single or doubly linked. In a standard linked list, each node has a reference to the next node in the list. In doubly linked lists each node has a reference to the next node as well as the previous node. This allows for the traversal of the nodes in both directions, making the data structure more flexible at the cost of more space required per node.

In addition, linked lists can have multiple links between nodes allowing for traversal in different orderings. These are commonly known as *multiply linked lists* and can be used for improving performance in certain conditions where it would be beneficial to have alternative traversals, at the cost of the space required for the extra pointers.

Linked lists can also be circularly linked where the tail points to the head making traversal circular. This allows for using it in instances where all nodes would be connected, for instance in describing polygons as linked polygonal chains. Having circularly linked lists poses a new set of problems as one could easily be stuck in an endless loop if not careful when traversing the list. This can be mitigated by adding a second pointer when traversing or making the head of the list unique and identifiable.

Some would argue that *sentinel nodes* should be used to determine when to terminate a search instead of using null values [29]. Sentinel nodes are nodes that do not contain any data relevant to the list, but rather are just used to make sure that a list always contains data. They can be used both as the head and tail of a list but are not recommended to use whenever you have many lists of small size as it increases the size overhead.

Table 2.3: Different types of spatiotemporal queries [3].

Query Type		Operation	Signature
Coordinate-based Queries		overlap, inside, etc.	$range \times \{segments\} \rightarrow \{segments\}$
Trajectory-based Queries	Topological Queries	enter, leave, cross, bypass, etc.	$range \times \{segments\} \rightarrow \{segments\}$
	Navigational Queries	traveled distance, covered area (top or average), speed, heading, parked	$\{segments\} \rightarrow int$ $\{segments\} \rightarrow real$ $\{segments\} \rightarrow bool$

2.4 Querying Spatial Data

The previously mentioned predicates defined by *The Open Geospatial Consortium* are called *The Dimensionally Extended 9-Intersection Model*, or *DE-9IM*, and include *intersects, disjoint, equals, touches, crosses, overlaps, contains* and *within*. These predicates can be used to create queries that can be used to extract data from objects' positions. In *Novel Approaches to the Indexing of Moving Object Trajectories* they group these into *coordinate-based queries* and *trajectory-based queries* [3]. The former includes common queries such as point, range and nearest-neighbor queries, and the latter includes topological queries such *enters, leave, cross* and *bypass* as well as navigational queries such as *speed, heading, parked*, etc. Here the authors describe the trajectory-based queries as "[...] very important, but also rather expensive [3, p. 397]." An overview of the different categorizations of queries can be seen in Table 2.3.

Topological queries do not only look at the spatial information of each data point but rather look at the spatial information that can be derived from looking at it in the context of a trajectory. The name comes from geospatial topology which is the study of the relationship between geographic objects. What makes topological queries so expensive to perform is that a spatial index such as an R-tree only indexes the spatial properties of each data point as a separate entity and not part of a topology. For instance, if one were to find all trajectories that start within a region one could perform a range search in the R-tree, but with the resulting points, one would be unable to tell just from their spatial properties whether or not they are the start of a trajectory as that is not a spatial property. This results in either having to use a separate index to do a lookup of each of the resulting data points to check whether or not it is the head of the trajectory, or performing a full table scan if no index is present. Performing an index scan is a more cost-effective option than conducting a full table scan, but it does require

additional storage space and still represents a non-trivial task in terms of processing time.

TB-trees solve this by making each leaf node only contain data points from one trajectory and then having pointers between all leaf nodes that contain data points from the same trajectory. This makes the spatial index also work as a trajectory index as one can traverse the leaf nodes that are relevant and answer queries such as whether or not the given trajectory started within the given region. The downside of this is that spatial queries that do not need topological information suffer as a result of worse splits causing larger MBRs and more overlap. This could be solved by having a traditional spatial indexing data structure alongside the TB-tree, but that would require more space and complexity.

2.5 Combining R-trees with linked lists

By combining R-trees with linked lists between nodes of the same trajectory, we can keep the benefits of high performance for range searches while still getting a performance boost for topological queries.

For instance, for the trajectory-based query, "Find all trajectories that start within region A and end outside region A", utilizing linked lists in the R-tree can improve the number of reads required to satisfy the query. When using either an iterative or indexed approach the only difference between them is how one identifies the heads and tails. The initial points are found by performing a range search on the tree with region A as the query window, resulting in a set of data points with potential trajectories. The data points are then filtered to remove any superfluous data points that have the same trajectory ID as any other candidate. In the iterative method, one would have to iterate over all the points in the dataset until one finds the first point with the correct trajectory ID. Then one would have to find the last point with the same ID and then check that the head is within the region and the tail is not. Worst case scenario the given trajectory is far down the list and one would have to iterate over almost the entire dataset before one finds a match for every single potential trajectory found in the range search. The index improves on this by allowing the iterating to start at the start of each potential trajectory reducing the search space down from the length of the dataset to, at maximum, the length of the longest trajectory.

If one instead uses links between the nodes of the same trajectory, there would be no need to access either the entire dataset or the index. Instead, in the initial filtering one starts by filtering out all data points that do not have their *prev* field set to null. This

removes any non-head data points, making sure that potential candidates that would not qualify in a later stage are removed early on. Then, one can traverse each trajectory directly until the current data point has the *next* field set to null to get the tails. Then, by filtering out all trajectories where the tail is not outside the region the process is finished. This significantly reduces the amount of data points that have to be checked both for their trajectory ID and spatial properties. This also translates into fewer pages read into memory, causing less IO overhead and better performance.

For storing the data on disk, using linked block allocation with data clustered on specific fields can yield benefits as opposed to contiguous allocation. For instance, if the average trajectory length is longer than the average number of unique trajectories located within a normal region, it would be beneficial to physically cluster the data on the trajectory ID as this would reduce the number of pages read during traversal. Conversely, if the number of unique trajectories within an average region is larger, it would be beneficial to store the data clustered on its spatial properties with data points close in physical space also being close on disk. These parameters are contingent on the dataset in addition to how the data is to be used. Another approach would be to use *mmap* which delegates the data retrieval to the operating system. However, this approach has been criticized in recent research papers for its hidden pitfalls. This, alongside several DBMSs stopping to use it in preference for traditional block buffer pools in recent years, may point to it not being the best solution [30].

2.5.1 Advantages

Creating explicit links between the nodes removes the need for an external index for effectively accessing all nodes in a given trajectory. Generating indexes takes both time and space with the index for the entire dataset used in this project taking up $\frac{326MB}{2440MB} \approx 13.3\%$ of the space of the original data when using PostgreSQL. Assuming that each reference to objects on the JVM takes up 4 bytes [31], and each node has two references each, the footprint of adding the links is approximately 267MB. The footprint when implemented in a database would be dependent on both the implementation and DBMS.

The most obvious advantage is the performance increase for topological queries. All queries that do not only require knowledge about the spatial properties but also the trajectory properties of the data should be able to see improvements to the query response time. As mentioned previously this is due to the amount of data points that have to be iterated over.

With a tree such as the TB-tree where the links are between leaf-nodes nodes, insert-

ing new data into the tree will require updating pointers whenever the leaf nodes are split. By having the links between the data points themselves inserting new nodes would not require updating pointers except for the previous and next node in case of inserting a node into an existing trajectory. As the links are not attached to the tree, no changes would need to be reflected in the tree itself. This also translates into that the implementation is agnostic to the underlying data structure. With pointers on the data directly, the spatial index can be swapped at any time depending on if the tree needs to be static or dynamic.

2.5.2 Disadvantages

Adding pointers between nodes in the dataset does require a migration of the existing tables to allow for using the pointers. However, as long as the ordering can be deduced by the DBMS it should be possible to automate this task. It would still require a lot of time to migrate large tables.

The method is not optimal when it comes to dynamic data as whenever new data is added to a trajectory, it would require updating the corresponding links. Insertion into linked lists has a constant time complexity, but that is not accounting for finding where the data should be inserted, which makes it linear. Most likely a trajectory will have data appended to the end, but there are use cases where it would be applicable to add a data point in the middle. Updating the trajectories is not something we have looked at in this project, but could be considered for future work.

As opposed to TB-trees, splitting based on the data's spatial properties instead of its trajectory segments does remove the benefit of having segments close together on physical storage. This can lead to decreased query performance on trajectory-based queries when implemented in a database as the data points may be unfavorably distributed across several pages. How this impacts performance could be another thesis in the future. Although not as effective as having each leaf node be a separate block, the physical storage can be configured in ways that benefit the data structure as previously mentioned above.

2.6 Related Work

In addition to the TB-trees mentioned earlier, other researchers are proposing data structures that seek to solve the same issues. One of these is the Riso-Tree, which "[...] partitions the graph into sub-graphs based on their connectivity to the spatial sub-regions"

[32]. This data structure leverages the power of graph databases to improve graph queries with spatial predicates (*GraSp* queries). The researchers are focusing on social graphs with spatial properties in the paper, but the data structure can be applied to the problems mentioned in this paper as well. The paper mentions *SpaIndex* and describes its approach in very brief words, which seems to somewhat match the method we are proposing. They describe the performance of this approach as "[...] exhibit unacceptable performance for applications that need to query the graph in real-time or near real-time" [32, p. 3]. A review of the literature on this topic indicates that, apart from papers authored by the same researchers, no other studies have mentioned *SpaIndex*. Despite being published in *ACM Transactions on Spatial Algorithms and Systems* in 2021, this paper has not gained much attention. This suggests that further investigation is necessary to evaluate the effectiveness of Riso-Tree and similar data structures in addressing the challenges of spatial graph querying.

Chapter 3

Method

Figure 3.1 shows the three different approaches to a topological query we will benchmark against each other. The query aims to find the tail of the trajectory T_5, which it obtained from a node as a result of a range search in the R-tree. This example is equivalent to a *leaves region* query, which we will discuss in more detail later. The topmost blocks show the iterative approach, which would be comparable to doing a full table scan in a database. Notice that, to find the tail, we first have to iterate through all the nodes that are not relevant to the T_5 trajectory. This results in having to iterate over 543 data points before it even checks the head of the trajectory. It also requires looking at one more node when the tail is reached, as it does not know that it is the tail before it has checked that the next data point has a different trajectory ID. The middle block shows an indexed approach, which would be equivalent to using an index in a database. The index here is an unclustered index that contains pointers to the heads of each trajectory. One could also have an index on the tails, making this query faster, but it would require twice the amount of space. In this example, we are using an index on the heads to assume the worst-case possible. Similar to the previous approach, we have to check the first node of the next trajectory to confirm that we have reached the tail of the trajectory, but in this case, we started the traversal at the head of the right trajectory. The bottommost blocks show an approach using linked lists. Here, the data points are modified to hold a pointer to the next and previous node in the trajectory. This allows for traversing the links until the next field is set to null to find the tail, or the prev field is set to null to find the head. As seen in the middle block, the head of T_5 is n_{543}, but the node found in our range search is $n_5 50$. This means that when looking for the tail, in the worst case, we are as close to the target node as the indexed method, but for all other cases, we are closer. In the example from the figure, we start seven nodes closer than the indexed method and we do not have to traverse beyond the desired trajectory to determine the tail.

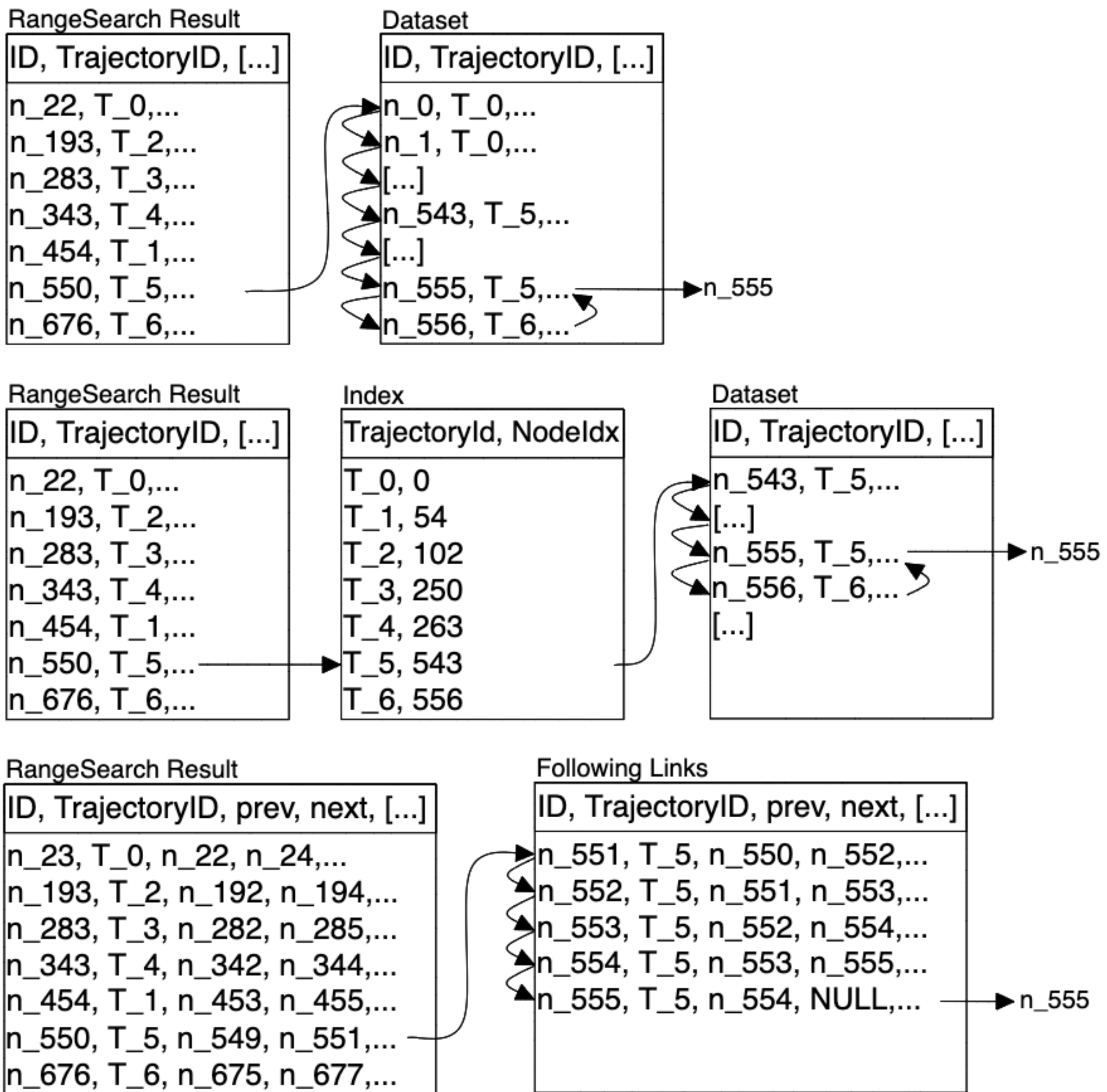


Figure 3.1: A visualization of the different approaches to finding the tail of a trajectory. The top shows an iterative approach, the middle shows a method with an index and the bottom shows a method with linked lists. Note how the different approaches have vastly different amounts of nodes visited.

3.1 Benchmarking Performance

To evaluate the performance of linked lists in topological queries, it is essential to ensure that the fewest possible factors are changed between methods. To ensure a fair comparison, the methods' implementations should be similar, with the only difference being the specific aspect under evaluation. Therefore, we chose to use an R-tree as the underlying data structure for the initial range search, which is easy to test if implemented properly. While it would be interesting to explore other underlying structures up against data structures that aim to improve topological queries, such investigation is beyond the scope of this thesis.

To test the performance of the algorithm, an implementation has been written in *Kotlin*, a language targeting the *Java Virtual Machine* (JVM) in the same way as Java does. *Kotlin* was chosen as the language of choice because it offers decent performance compared to languages like *Python* while being faster to develop in and more high-level than languages like *C++*. The reason for choosing Kotlin over Java is that the former offers faster development through less boilerplate and verbosity and an increase of syntactic sugar. The author's proficiency in the language was also a determining factor.

Implementing the algorithm in a higher-level programming language does present some drawbacks. Mainly, the speed of the searches will not match real-world usage, primarily due to how a real DBMS optimizes queries in advanced ways, but also the larger overhead caused by the virtual machine used by Kotlin. It should be noted that there are databases, such as *Cassandra*[1], that are written in Java which targets the JVM. This shows that the data generated in this project could somewhat mimic real-life performance in some cases, keeping in mind that these databases are highly optimized.

Frameworks such as *ELKI*[2] were considered due to their existing data structures, benchmarking, and visualization capabilities, but were discarded due to the amount of work that would be needed to become proficient in the framework. Instead, an R-tree implementation was written from the ground up, including a benchmarking tool that allows for timing query response time. The R-tree has some key features missing from being complete, with the main one being that removing nodes are not supported after the initial tree has been built. This was done to improve development speed and was deemed unnecessary due to the scope of this project.

[1] `https://cassandra.apache.org/_/index.html`

[2] `https://elki-project.github.io/`

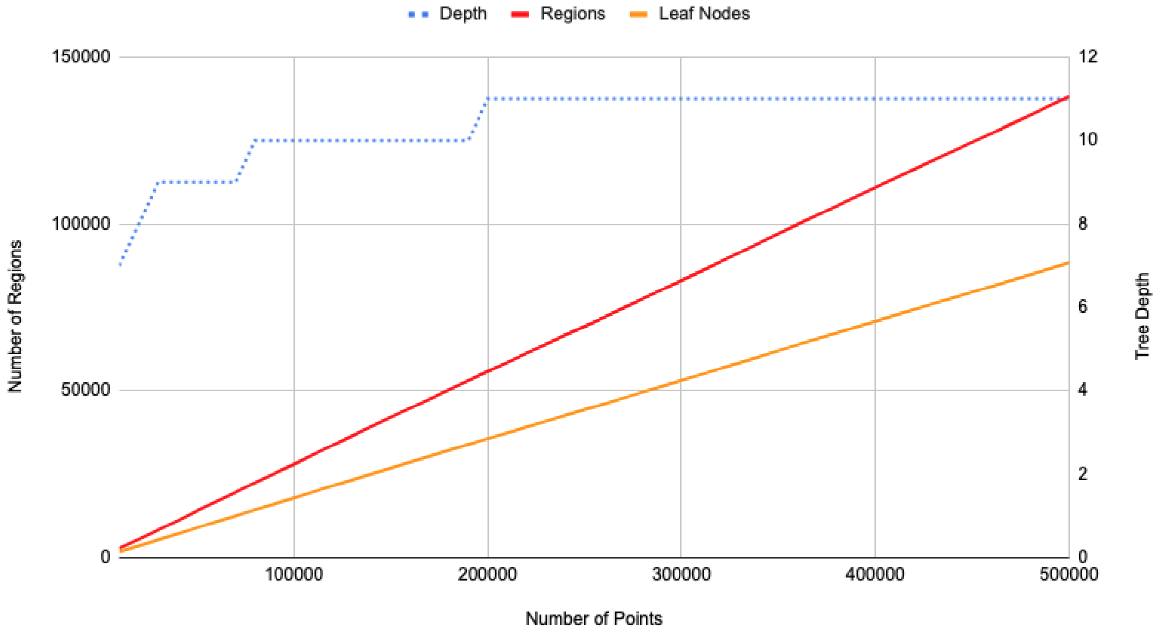


Figure 3.2: Number of nodes and leaf nodes in the R-tree for all data points up to 500 000.

3.2 R-tree Parameters

The choice of parameters affects how the tree performs. When benchmarking, it is important to set these parameters to reasonable values that ensure the tree functions as intended. While good tree performance is essential in real-world scenarios, for benchmarking topological queries, the tree only needs to function sufficiently, not optimally. This led to the adjustment of several parameters to enhance tree build performance and development speed.

The max filling factor, which is the maximum amount of data points a leaf node can contain, was set to nine as it was found to give decent performance. For reference, for 500 000 data points, the average trajectory contains $\approx$ 49 nodes, meaning the average trajectory would have to be split across at least five leaf nodes without taking the average fill grade into account.

The minimum filling grade in an R-tree is usually recommended to be around 40% of the maximum filling factor. $9 \times 0.4 = 3.6$ and to make it a round number seeing as data points cannot be split, it was floored, making the minimum filling factor three. This means that any leaf node cannot contain less than three data points. Whenever a split, resulting in leaf nodes l_1 and l_2, has distributed x data points such that the remaining

points to be distributed, y, is equal to $minFill - \{l_2\}$, meaning that n_2 needs the remaining points to reach the minimum filling grade, they will be assigned to l_2 disregarding if it is the optimal solution as described in [10]. The average fill grade stays consistent at around 62% for all different values of points, making the average leaf node have 5.62 data points within it. This is a lower average value than what a normal R-tree has, which is around 70% of the capacity [10]. There can be several factors that contribute to this, such as using a linear splitting algorithm or how the data points are distributed.

The fanout of the tree determines how many children each non-leaf node can contain. Whenever the fanout is exceeded, the node would have to be split. For instance, with a fanout of one, each node can only contain one child, making the tree linear. With a fanout of two, the tree would be a binary tree. Some developers recommended using 2^d for the fanout of R-trees where d is the number of dimensions of the data. For instance, a two-dimensional R-tree would, in this case, have a fanout of $2^2 = 4$, and a three-dimensional R-tree would have a fanout of $2^3 = 8$. This formula has not been scientifically proven efficient but has some developers swearing by it as long as the tree is only in memory. For this reason, as the data is only in two dimensions, a fanout of four was selected.

As the goal of this benchmark is not to test how well the tree performs, but rather how topological queries perform with and without links, the splitting algorithm is not crucial. As we use the same splitting algorithm for both the queries with and without links, it does not matter what algorithm is used. Because of this, it was decided to use a linear splitting algorithm as opposed to quadratic, exponential or other algorithms. This does result in worse query performance on the tree, but it speeds up development time and the time it takes for the tree to build. In Figure 3.2 there is a graph depicting the number of leaf nodes compared to the total amount of nodes and the corresponding tree depth.

3.2.1 Special considerations

During development, there were some special considerations taken into account to ensure that the tree is working as intended. These result in the R-tree behaving slightly differently from other implementations, but the core principle is the same, and the performance should be minimally affected.

Early implementations of the tree had the MBR size calculated using the area of the MBR. The size is used in the splitting algorithm to determine which region each node should be added to ensure the smallest enlargement possible. When using the area, the size is calculated using the formula $A = width \times length$. This would be fine if it were

Table 3.1: Summary of R-tree Parameters

Parameter	**Value**	**Explanation**
Max Filling Factor	9	Maximum number of data points a leaf node can contain. Decided upon as it provides decent performance.
Min Filling Grade	3	Set to around 40% of the max filling factor. Any leaf node cannot contain less than this number of data points.
Average Fill Grade	62%	Average percentage of a leaf node's capacity that is filled. This is lower than the typical average for an R-tree, which is around 70%.
Fanout	4	Number of children each non-leaf node can contain. Set to 2^d, where d is the number of dimensions of the data (in this case, 2).
Splitting Algorithm	Linear	Chosen as it speeds up development time and tree build time, even though it may result in worse query performance.

not for the fact that each node is a single point, meaning that they have an effective width and height of zero. This means that if two points were at the same X or Y value, you would get a width or height value of zero, meaning that the calculated area would be zero. This led to some interesting rectangles spanning a large area horizontally with zero height. To mitigate this, the formula was exchanged for the circumference formula $C = 2 \times length + 2 \times width$ to ensure that the nodes would be placed in the appropriate regions. After testing, it seems as though this does not pose any unintended side effects when splitting. For real-world data using GPS coordinates, this is quite unlikely to happen, but having the edge case removed is always beneficial. An example of how this affects the splits can be seen in Figure 3.3. Another way to remedy this could be to add padding to the initial MBR for each point, making the MBR height and width non-zero.

In Kotlin, there is a pre-defined class called *LinkedList*[3] which it inherits from Java. This provides an implementation of a doubly linked list that can be used out of the box. It allows for accessing and manipulating the list in several different ways. For this project, it was deemed unnecessary to use this class, as the list would always be accessed through the nodes instead of through a reference to the list. The linked lists are instead created by having links between the data points without any overlying class that provides functionality.

[3]`https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html`

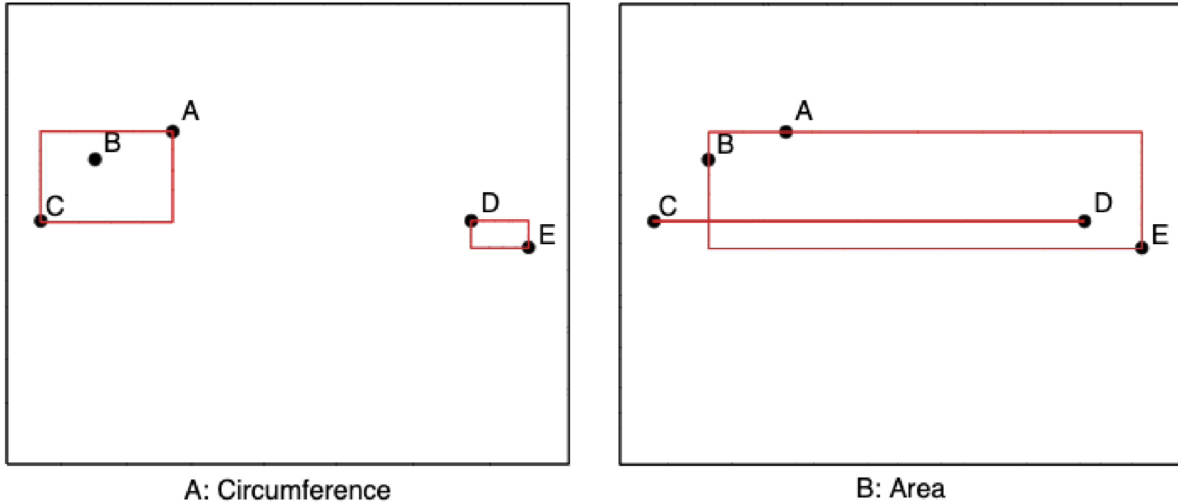


Figure 3.3: Example of how using area vs circumference affects the splitting when calculating the MBR when C and D have the same Y-value and their height is zero.

3.3 Choice of Dataset

There are quite a few different datasets publicly available that would be suitable for this project. To determine which datasets were relevant, we needed to consider various aspects to ensure they met our requirements. These aspects include:

- **Spatial properties.** For the dataset to be relevant, it must have spatial properties. Either GPS coordinates or something similar that will distribute the data points in an (X, Y) grid.
- **Be based on real-world data.** Real-world data makes the data points more genuine and realistic. Generating a new dataset could also be used, but that might not take into account the unexpected factors one would encounter with real-world data.
- **Some kind of order.** This can either be through having links between data points, implicit ordering through references such as IDs, or inductive ordering through the ordering of the rows or timestamps.
- **Sufficient data.** A dataset with only a few data points would not be sufficient, as the search times would be indistinguishable from each other if they take too short

a time to complete. Ensuring that there are enough data points to get an accurate representation of the performance differences is important.

Many large tech companies depend on spatial data. Companies such as Strava have built empires using, aggregating, manipulating, and mining spatial data. Initially, Strava was considered in case they had any anonymized datasets available. Being the largest fitness tracking app in the world, one would assume that they have a lot of publicly available data. Unfortunately, not in a way that was suitable for generating a large dataset. To access their data, one would have to use their open API, but it would require explicit consent from all users who have generated the data. In addition, Strava does impose quite a few restrictions on how the dataset is used and distributed. These restrictions posed too much of a hassle to make it a viable option.

Kaggle is a community for data science and machine learning. They regularly host competitions where users are provided a dataset and compete to create the best algorithm to solve a problem. These datasets are not created by Kaggle themselves, but rather provided by companies, users, or researchers. Throughout Kaggle's history, there have been several competitions, and as a result, there are many datasets publicly available for download. As these are used for different competitions, they consist of all types of data and sizes. Some datasets are only a few kilobytes in size, while others are several gigabytes. Fortunately, they have a search functionality that allows users to filter datasets based on several different factors, including size, date, data type, and others. Section 3.3 shows some of the datasets that were considered, both from Kaggle and other sources. In the end, the Taxi Trajectory Prediction dataset was chosen, which consists of taxi trips taken in the city of Porto, Portugal within a timeframe. It fulfilled all the requirements while also being descriptive and easy to understand. This dataset was also recommended by Svein Erik Bratsberg, as it has been used in previous research projects surrounding spatial data and trajectories.

The dataset was modified to be more suitable for insertion into both the program and a database. In its initial state, the dataset would require unnecessary complex parsing to be used in the program. This was due to each trajectory being a row with each data point being defined in a polyline field which was stored as a string. A script was written to convert the dataset into a more easily parsed format where each data point has its own row with *LATITUDE* and *LONGITUDE* fields in place of the *POLYLINE* field. This does increase the file size, but this is not an issue. The order of the data points was maintained by making sure that each row was inserted in the original order of the data points. Each row was also assigned its corresponding trajectory ID to make sure that that information is preserved. To make sure that there were no parsing errors the script

Table 3.2: Datasets that were considered, both from Kaggle and other sources.

Name	**Spatial Properties**	**Real-World Data**	**Ordering**	**Sufficient Data**
Microsoft Geolife GPS Trajectory Dataset	Yes	Yes	Timestamped	Yes - 1.67 GB
T-Drive Trajectory Dataset	Yes	Yes	Timestamped	Yes - 803 MB
GPS data from Rio de Janeiro buses	Yes	Yes	Timestamped	Yes - 6 GB
Wrocław public transport	Yes	Yes	Timestamped	Yes - 3.56 GB
ECML /PKDD 15: Taxi Trajectory Prediction	Yes	Yes	Polyline	Yes - 533.7 MB (compressed)

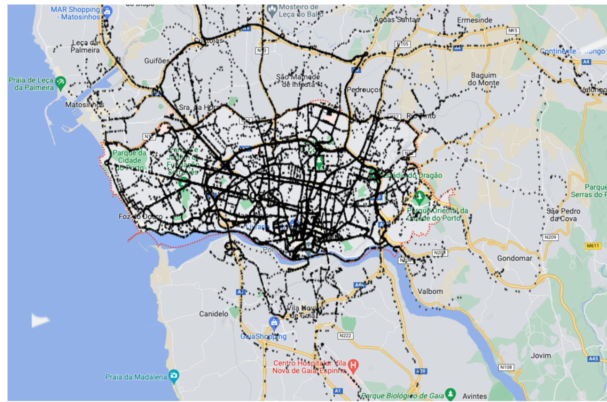


Figure 3.4: The streets of Porto as visualized by 50 000 data points from the Porto Taxi Trajectory Prediction dataset overlaid a screenshot of Google Maps.

also made sure to remove any trajectories that were empty or had vital data missing. It also removed redundant fields such as *MISSING_DATA, CALL_TYPE, ORIGIN_CALL, ORIGIN_STAND, DAY_TYPE,* and *TIMESTAMP*. The code for this script is available in Appendix B.

3.4 Benchmarking

To determine whether or not the links improve performance on topological queries, they need to be benchmarked against each other to evaluate their performance. It was decided to compare against both an iterative method as well as an indexed version to give more perspective on how big of an impact this can have. When someone stores data in a

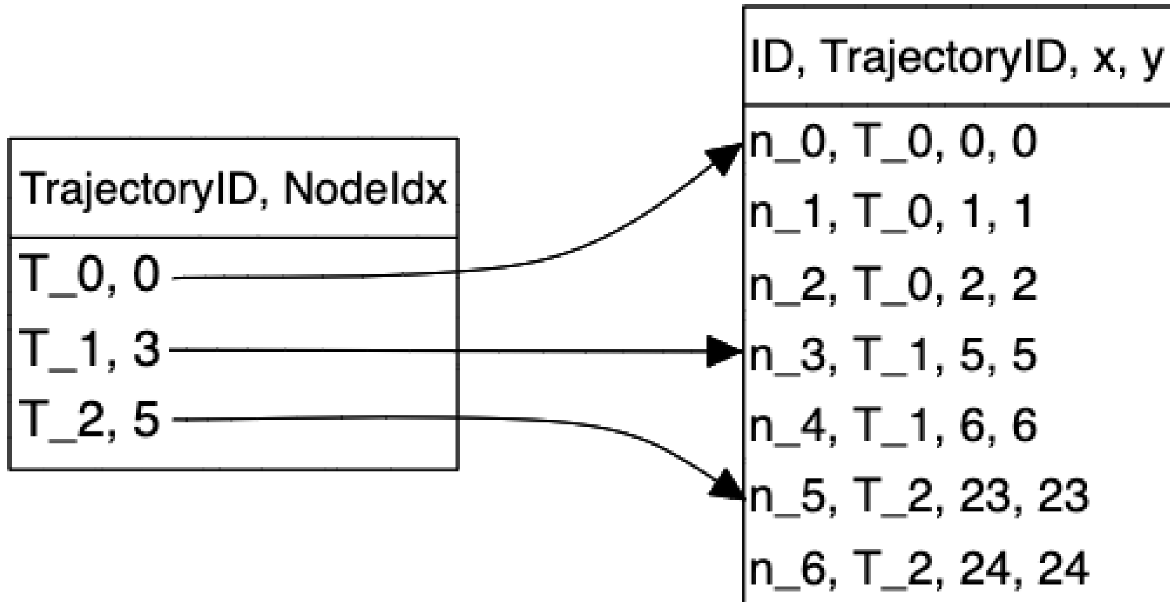


Figure 3.5: The index used in benchmarking visualized. The index on the left contains pointers to the position of the first data point of each trajectory within the list of points.

database, it is uncommon not to utilize an index to ensure that queries are sped up. To simulate this, an index has been created using a hashmap that links the trajectory ID to the index of the first data point in each trajectory within the list of points, functioning as an unclustered index.

3.4.1 Number of data points

Having a large number of data points allows for obtaining an accurate representation of the performance differences between utilizing linked lists and not. During testing, it was found that using fewer than ten thousand data points would make the performance difference almost indiscernible, as expected. By creating a range of data points to benchmark, we can ensure that the trends will be highlighted, and the performance can be compared at several levels. It was decided to use a range from 10 000 data points up to 500 000 with an increment of 10 000 per benchmark. This range was chosen as it provides a large range of data points while still ensuring that the process does not consume too much time.

3.5 Queries

Based on the topological queries mentioned in Section 2.4, the following queries are to be used in the benchmark: *Start and End Search, Crosses Search, Enters Search,* and *Leaves Search.* A visualization of some trajectories that satisfy these queries can be seen

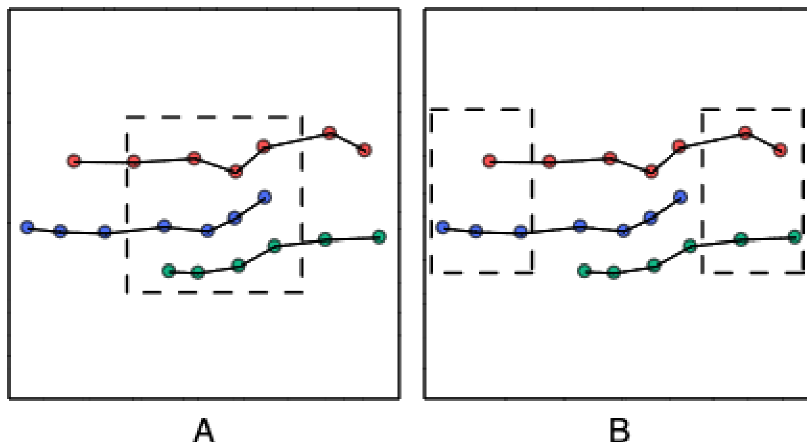


Figure 3.6: The different topological queries. Figure A shows three different trajectories that will satisfy three different queries. The red (top) satisfies the crosses query, the blue (middle) trajectory satisfies the enters query and the green (bottom) satisfies the leaves query. Note that each of the trajectories only satisfies one of the queries. Figure B shows one trajectory (red) satisfies the start and end query while the others do not.

in Figure 3.6.

3.5.1 Start and End Query

The *Start and End* query accepts two query windows W_1 and W_2 and returns all trajectories in which the head of the trajectory is located within W_1 and the tail is located within W_2. Figure 3.7 depicts W_1 and W_2 as red and blue rectangles respectively.

For all approaches, the first step to this query involves performing a range search on the tree for one of the two regions. From the resulting data points, one can filter out any superfluous nodes characterized by duplicate trajectory IDs. As briefly mentioned in Section 2.5 in the iterative approach, a full table scan is required for each node to ascertain if the given node is the head of the trajectory. This process is repeated for the second region, but in this instance, the tails are identified. Finally, an intersection of the resulting trajectory IDs from the first and second regions is returned. The costly parts of the iterative approach for this query are:

- Range search the tree for both W_1 and W_2: $O(2 \times log(N + k))$ where n is the total number of data points and k is the number of data points within the given region.
- Filtering nodes with unique trajectory ID: $O(k)$.
- Iterating over all points to find the head of each trajectory ID: $O(N \times k)$.

- Iterating over all the points to find the tail of each trajectory ID: $O(N \times k)$.
- Returning the intersection of the heads and tails: $O(k^2)$.

This results in a worst-case time complexity of $O(log(N)+k)+O(k)+O(N\times k)+O(n\times k)+O(k^2)$. However, as they are all dominated by $O(N \times k)$ the time complexity can be simplified to $O(N \times k)$.

When using an index approach instead of performing the full table scans one can instead perform an index scan, a performance increase over using table scans. Assuming the same as above this comes out with a worst-case time complexity of $O(k \times N)$ as the time complexity of the index scan is $O(k\times N)$ when the entire dataset is one trajectory.

With a linked list approach the query can be optimized further. Instead of performing two range searches, only one has to be performed due to the tails being given by traversing the links. This leads to a worst-case time complexity of $O(N+k)+O(k)+O(N\times k)+O(1)$ which is dominated by $O(N \times k)$. When simplified this is equal to the other methods. However, this is assuming the absolute worst-case scenario where the entire dataset is one long trajectory, making the method have to traverse the entire dataset. Using real-life data this should not be the case and the approach should be significantly faster. It should also be noted that when simplifying, although a common practice, one does lose information on time complexity as very large factors can be dominated, making several algorithms look equally performant on paper when the real-life performance is vastly different. Assuming that the average node is halfway through a trajectory this would require only traversing half the trajectory in each direction for both finding the head and the tail. Comparing this to when using an index, finding the tail does not necessarily require traversing the entire trajectory. The linked list approach also ignores the intersection step that the two other approaches require, which should further improve relative performance.

3.5.2 Crosses Query

This query finds all trajectories where either the head or tail is located within the query window W but does have at least one data point contained within it. Although this search uses one range search less than the previous query it is still quite computationally expensive when dealing with large amounts of data. When using an iterative implementation this requires first doing a range search on the tree for all data points within region W, filtering out duplicate trajectory IDs, then iterating over all points to check that their heads and tails are not contained within the region. This requires at least one full table scan for each data point, assuming that there is no temporary storage of heads and tails

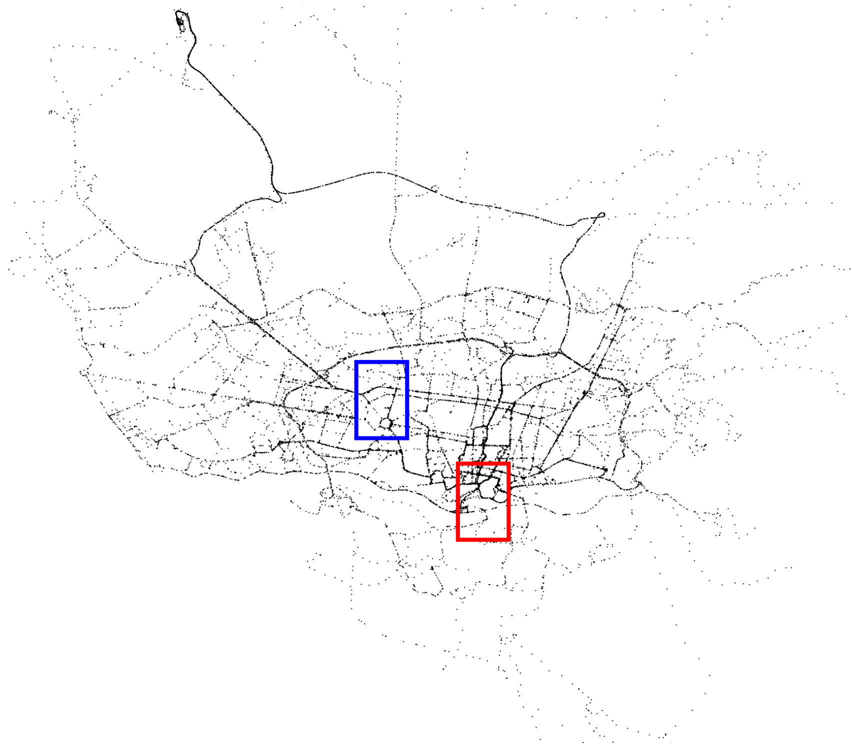


Figure 3.7: W_1 and W_2 used in the benchmarks visualized. The start region (W_1) is highlighted in red (bottom right), and the end region (W_2) is highlighted in blue (top left).

between nodes. The indexed method replaces the table scans with index scans, making identifying heads constant and finding the tails only requires iterating over each respective trajectory. When using the linked list approach one can traverse the links to find the heads and tails and then do a check against the query window's constraints in constant time. As with the previous method, the simplified worst-time complexities are the same for all methods, but the performance difference should be noticeable.

3.5.3 Enters and Leaves Query

The queries for trajectories entering and leaving regions both take a region W and find all trajectories either going into the region and not exiting or all trajectories that start in a region and end outside the region respectively. This gives them both the same time complexity and number of data points visited. It should be noted that these are not *strictly enters* or *strictly leaves*, which would take into account that the trajectories that satisfy the queries should not be able to re-enter or leave a region multiple times.

For an iterative implementation, this would entail a range search on region W, filtering out all duplicate trajectory IDs, traversing all the points to filter out all the data points that are not a start or an end to a trajectory, and then finding the opposing end making

sure that it is outside the query window. When using an index the only change is that the table scans are replaced by index lookups. For the linked list method the lookup is replaced by traversing the links until one end and then to the other end, making sure that the respective ends are either within the region or outside the region, depending on the query.

3.6 Environment

The benchmarks were conducted on a 16-inch MacBook Pro 2019 edition with a 2.6GHz 6-core Intel Core i7 processor with 32 GB of 2667 MHz DDR4 RAM running macOS Monterey 12.6. The benchmarks were run with no other applications open and the computer was left alone for the entirety of each benchmark. Each query was run 10 times concurrently using coroutines. The use of coroutines to run queries concurrently may affect query times, but from testing, this does not seem to be a problem. Running each query ten times was meant to reduce outliers, but as we can see below it was not the case.

3.7 Preventing Anomalies

Early benchmarks contained outliers in terms of runtime for the different queries. When running several times it was obvious that the outliers were unrelated to the number of data points, meaning that they were not dependent on the code or dataset, but rather anomalies that were outside our control. The JVM is a complicated system with many moving parts that can impact performance. Features such as garbage collection can impact the runtime without warning. To combat this, each run, containing ten concurrent runs of all the data points, was run ten times to remove the outliers. Although a few outliers remain in the graphs, they are much fewer and less severe than the previous ones which were up to ten times slower than expected. In addition to these spikes in performance, the first few runs, from 1000 to 10 000 data points, were slower than the later runs with more data points. This can be potentially be attributed to the fact that the JVM uses Just-in-Time (JIT) compilation to optimize the code. This does take some time, and starting benchmarking from the very start of the program can cause a cold-start problem where the optimization is done yet, making the performance worse for the first few results. By having the program first run for the first 50 000 points before starting benchmarking the performance of the first few fell in line with expected values.

3.8 Reducing Search Space

Early implementations showed that the benchmarks took way too long, making getting enough data unfeasible. This was largely caused by the regions being too big, making the search space contain several thousand data points. To mitigate this, the regions were made smaller, improving the performance while still containing enough data. In addition, queries such as start and end query proved to be significantly faster than others. This was caused by the two regions not having any overlap in trajectories, making the most costly operations get skipped as there were no points to iterate over. By moving the end region to another place where it is guaranteed to have trajectories in common with the start region for all used number of data points this was mitigated.

As the queries return a list containing the IDs of the trajectories that match the predicate, a way to reduce the search space earlier was to only consider one data point for each trajectory within the region. This way, if there are several data points from the same trajectory being returned by the range search, one can safely only check one of them. This was done using the built-in array function *find* which returns the first element that satisfies the query. The safety is dependent on the result list from the range search being in the correct order, which it should be if the initial dataset is in the correct order and the tree is balanced, which the R-tree is.

All of these measures combined proved to be an effective way to improve query performances. The first benchmarks took around 27 hours to complete each time, but after reducing the search space it only took around one hour to complete them all.

Chapter 4

Results

From the benchmarks, it is clear that utilizing linked lists is more effective than iterating through every data point, and even more effective than using a traditional index. Looking at Figure 4.1 we can see that every query using linked lists outperforms the other methods by a significant margin. Our charts present the response time of the queries instead of the throughput of the system when using this method. This is because the iterative approach has such a slow response time that presenting them as throughput would be counter-intuitive. The iterative approach is slowest for all queries, as expected, but the fact that using linked lists is consistently faster than using an index is interesting. This can potentially be attributed to the fact that the index always starts at the start of the trajectory, but the linked list approach can start wherever making the search space smaller whenever the initial data point is not the head of the trajectory. There can be other factors such as how the compiler compiles linked lists and arrays differently.

Interestingly, looking at Figure 4.3 we can see that when using an index the start and end search underperforms compared to the other queries, which is the opposite of what can be seen for the linked list and iterative methods. The higher query time when using an index can be attributed to the larger search space as a result of having two regions, causing more index scans. What is weird is that this should also be the case for the other methods. At first, one missing filtration of duplicate trajectories in one of the regions was thought to be the case for this result, but after fixing that error the search is still slower than the other indexed queries. Although the start and end search is the slowest of the indexed queries, if the approach were to be doubly indexed we could potentially see speeds faster than the other searches, potentially matching or bypassing the speeds of the linked list approach. This would, however, require another index of the same size as the first one, making it take up even more space.

When looking at the different figures it is important to keep in mind the scale of each

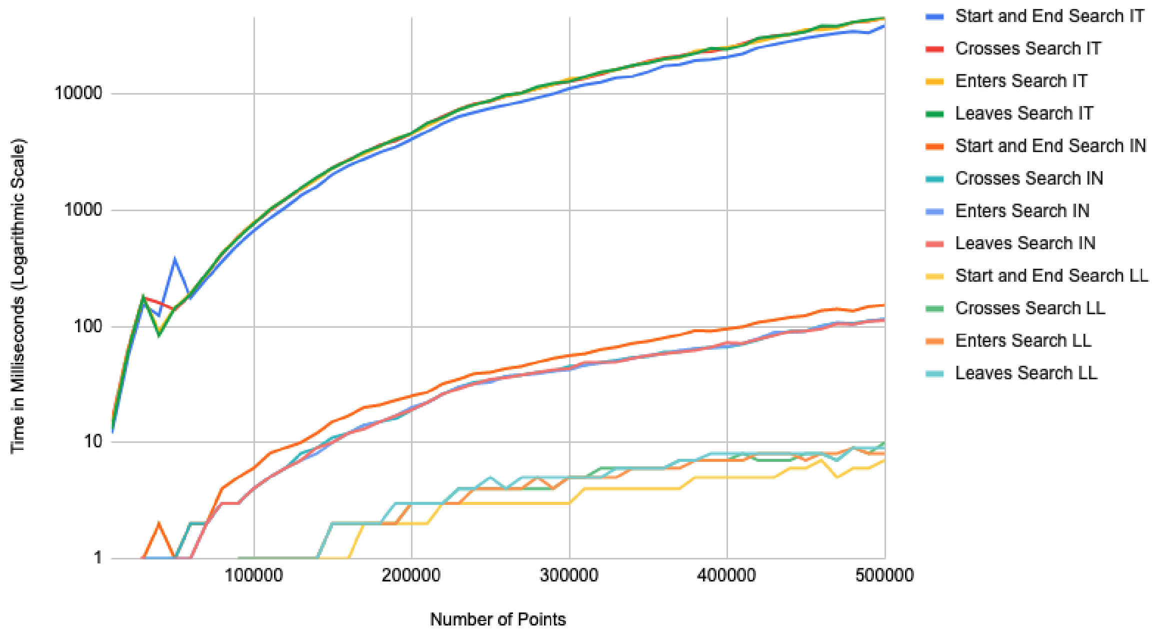


Figure 4.1: Median time spent on each query for all queries with a logarithmic scale.

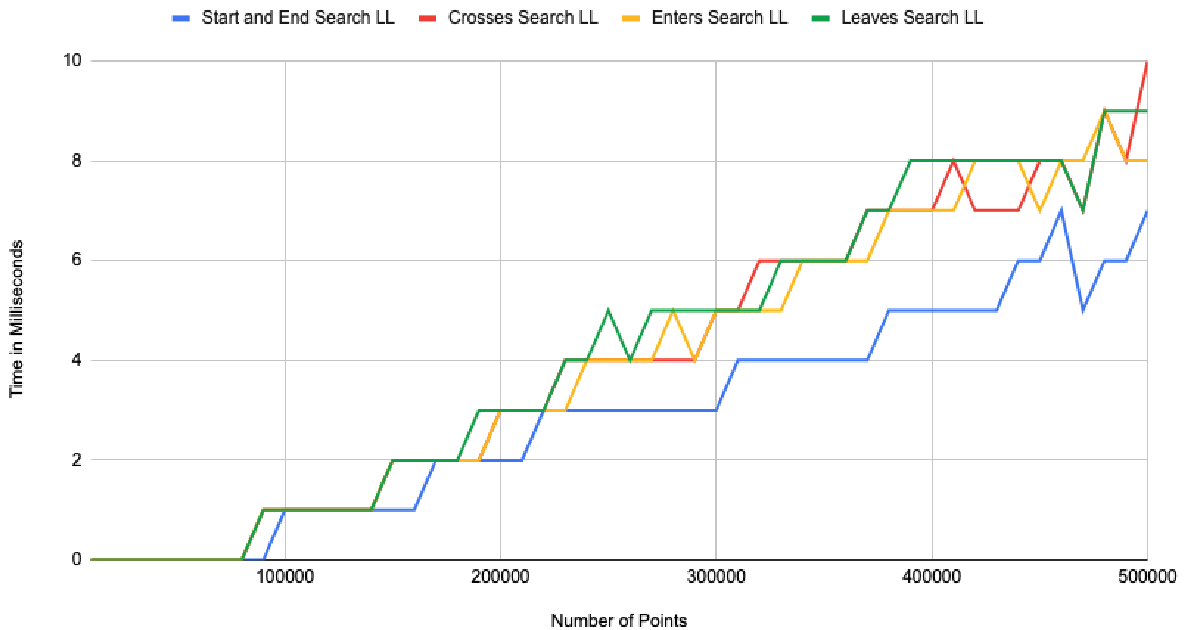


Figure 4.2: Median time spent on each query in milliseconds for queries with links.

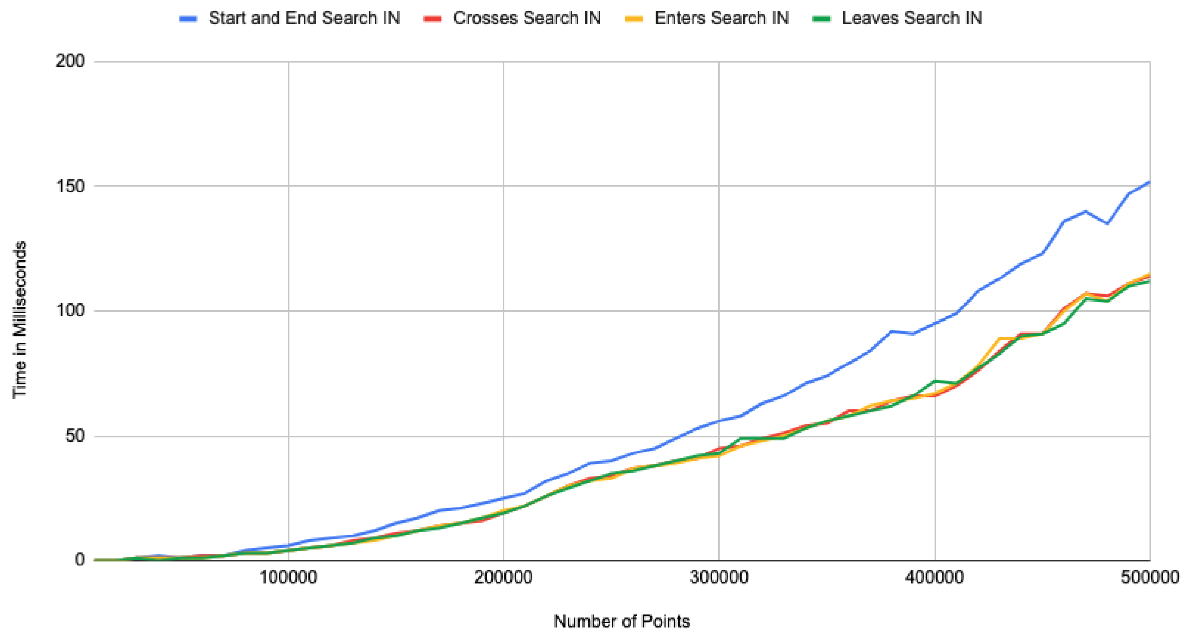


Figure 4.3: Median time spent on each query in milliseconds for queries with an index.

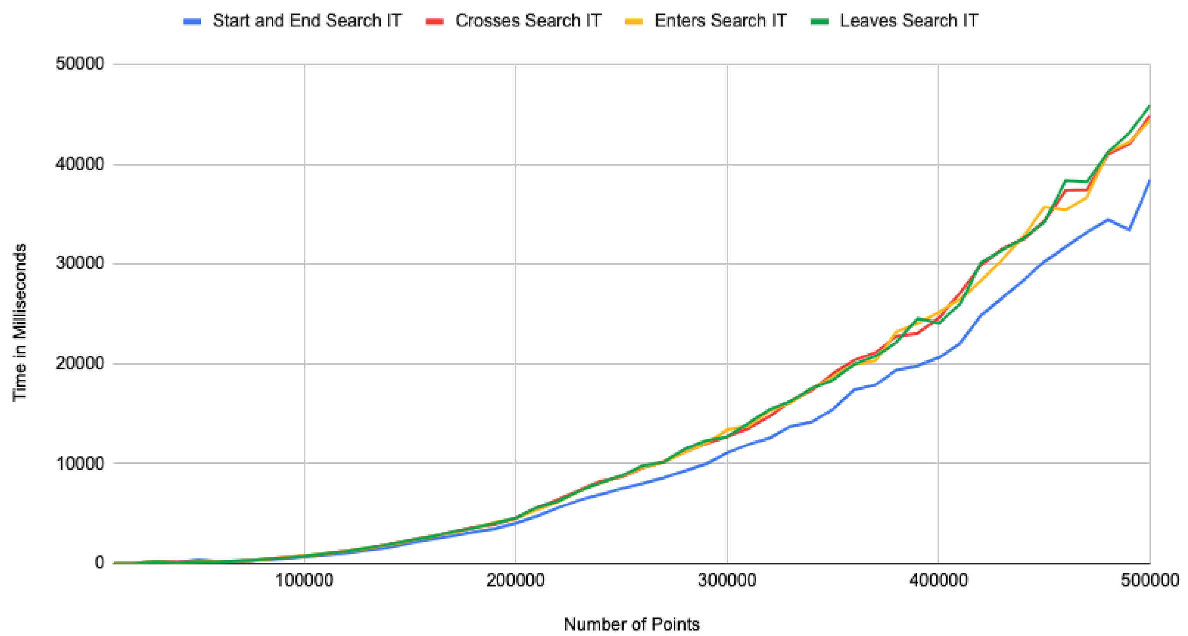


Figure 4.4: Median query times using iterative search.

one. Figure 4.2 may look like the approach is very unpredictable with high variability, but when working with times less than ten milliseconds it is expected that the graph will be more unstable than a graph with a ten times larger scale. A few milliseconds to or fro can be attributed to background processes or operating system factors that cannot be controlled in the testing environment.

Additional graphs alongside the full dataset are located in the appendices.

4.1 Reproducability

The complete codebase for this research project is publicly available via GitHub[1] and the dataset is available at Kaggle[2]. All the data produced is publically available on Google Sheets[3] The code itself should be usable without any configuration, assuming one has Java installed, but the dataset does require some pre-processing to be compatible with the program. The code for converting the dataset, as well as a description of how it was manipulated, is in Section 3.3. By sharing our work through this platform, we hope that this will allow others to try the method out themselves and improve on it if possible.

With our codebase, it is possible to reproduce the results of our study rigorously, allowing other researchers to validate and extend our findings. The instructions provided in the documentation should facilitate a straightforward setup, and we encourage other researchers to explore our work and develop new implementations to suit their specific needs. However, we must again note that the query speeds obtained through our implementation are dependent on several factors, including hardware, operating system, and programming language, which can affect the results of benchmarking. Thus, it is essential to exercise caution when interpreting our findings, and we encourage researchers to consider these factors when conducting their experiments.

Hopefully, the availability of our open-source codebase will be a useful resource for the academic community, enabling other researchers to build upon and extend our work to advance the field of spatial data further.

[1] `https://github.com/erikskaar/master`

[2] `https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i/data`

[3] `https://docs.google.com/spreadsheets/d/1uQ8rgErHMeXFl59MyxhLdcDMau2iGkOQXdjrnHVskZY/edit?usp=sharing`

Chapter 5

Discussion

Even though the results show that using linked lists outperforms using an index for topological queries, it is important to note that these results are mainly theoretical, meaning that if this were to be implemented in a database management system the results would look quite different. In all database management systems, the query optimizer and planner have a lot of ways to improve queries as much as possible, making them faster than one would expect. Therefore, a naive implementation of this approach would potentially lead to disappointing results without the proper implementation into the DBMS.

For this dataset the average trajectory contains around 49 data points, meaning that the trajectories are quite short. The length of the trajectories can affect performance significantly, but based on the results here it should be safe to assume that using linked lists should perform better independent of the trajectory length due to how it limits the search space.

The reason why the start and end query performs worse than other queries when using the indexed approach as opposed to the other approaches remains a mystery. The most likely candidate is that there has been an implementation error that the tests written do not cover. In addition to this, the iterative approach is still unstable for a low amount of data points, which is most likely caused by too little warm-up for the JVM. However, during testing, no matter how much warm-up provided the graph would not smooth out without kinks. Testing with other datasets or different amounts of data points could provide insight if this is an implementation problem or a data problem.

The work for this thesis has been based on the topological queries listed in the TB-trees proposal paper in addition to the start and end search [3]. These are only a few queries that use some of the nine intersection predicates in DE-9IM. Because of this, for other queries using different predicates, the results can give other outcomes. Queries such

as *bypasses* are more complex and will use much larger search spaces than the queries that have been used here. This can alter the relative performance of our method compared to established methods.

As mentioned in Section 2.6, the researchers of the Riso-Tree paper say that *SpaIndex* (read: similar method) does not exhibit the performance needed to perform queries in real-time or near real-time. However, looking at our results this can be disputed. Although our implementation is implemented in a language and ecosystem which is known to not be the fastest, the data shows that performing a complex query only takes as long as performing a simple R-tree range search (by deducting the range search time from the query response time) [33]. It should be noted that the R-tree implementation is not perfect and could be improved, but having a topological query on 500 000 data points takes less than 10 milliseconds when performed in a JVM language points to there being potential for near real-time query performance possible with the right implementation and programming language.

Looking at the comparison methods there are ways to optimize them to improve performance. For instance, when iterating over all the data points in the dataset in the iterative approach one could cache all the heads and tails during the first iteration. This would save the remaining results from the range search from having to do the same thing repeatedly. The start and end query can be optimized more for the three different methods, assuming that the dataset is large enough that the range search is trivial compared to iterating over the relevant data using the respective method. This can be done by performing two range searches, one for each region, and using the region with the smallest number of unique trajectories as the base for traversing. How big the dataset needs to be depends on the method used. Looking at Figure 5.1, for 500 000 data points the range search will take approximately 5 milliseconds, which is a significant amount when using the linked list method, but for the iterative method, this is insignificant making it a worthwhile trade. For the indexed approach, one could add a secondary index, for instance indexing the tails. This would speed up queries that need to find the tail of the trajectory.

For the queries used in these experiments when using linked lists, one could improve the query speed by changing the pointers from pointing to the next and previous nodes and instead only pointing them to the head and tails. This is due to how our queries only take into account the positions of the heads and tails, instead of looking at intermittent nodes. This would not work for other queries such as *strictly enters* or *strictly leaving*, as they require checking all the nodes in the trajectories. This shows that there are further optimizations possible for the linked list approach as well, depending on the dataset and

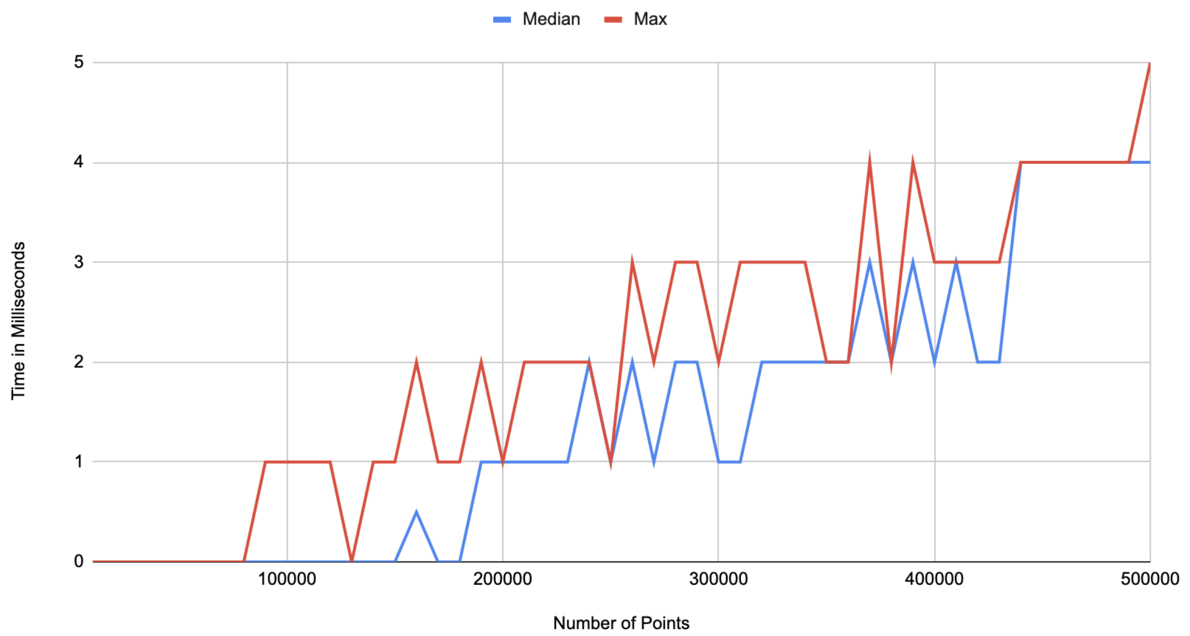


Figure 5.1: Max and median time spent on each range search for all points in milliseconds.

queries.

Chapter 6

Conclusion

Throughout this paper, we have undertaken an extensive investigation of improving topological queries by leveraging the power of linked lists. Our empirical results have demonstrated that this approach can be a viable and effective alternative to the existing solutions in use today. While our analysis is purely theoretical, limited only to a specific dataset with particular queries, our findings are undoubtedly promising, suggesting that linked lists can indeed offer a compelling solution to enhance topological query speeds without compromising spatial query performance.

6.1 Future Work

Looking toward the future, further testing of the proposed method with different datasets and larger amounts of data could prove valuable in assessing its scalability and applicability. A comparative study with related data structures, such as TB-trees, would also be insightful in understanding the relative strengths and limitations of each approach. The ultimate factor determining the superiority of the linked list approach over existing alternatives would depend on the range search improvement achieved by the underlying structure as they could offset the performance gained from bundling trajectory segments together in leaf nodes and storing trajectories physically close on disk.

Moreover, exploring the optimal implementation of the linked list approach into a database management system could be a fascinating avenue for further research. One possible approach could involve leveraging recursive queries and utilizing the *next* and *prev* fields to traverse the trajectory, as implemented in this project. An alternative implementation could involve using graph databases to traverse the trajectories. This could be achieved by creating the underlying data structure as a clustered ad-hoc tree using the

graph functionality or by having the underlying structure remain unclustered in the same manner as a relational database system. Regardless of the approach taken, traversing the trajectories using a graph database management system could potentially lead to even faster performance than when using a traditional relational database management system. Further research in this direction could prove valuable in the development of more efficient and scalable approaches for managing large datasets. This would require finding an efficient and suitable way to organize the data on disk to reduce the number of pages required to read when traversing the trajectories.

In Chapter 4 we mentioned how all the methods could be further improved. In the future, adapting the linked list approach to different datasets of different types and sizes, as well as different queries can further uncover the strengths and weaknesses of the approach.

As briefly mentioned in Subsection 2.5.2, looking into how the implementation handles the insertion of data connected to established trajectories could be something to look into. In theory, it is only as complex as inserting into the linked list and inserting into the underlying data structure, but there could be some additional factors to consider, such as the order of the dataset and how it affects the implementation of the queries. The performance of insertion can be the determining factor if the method is suitable for handling data in real-time, which is a common use case when dealing with trajectories and spatial data.

6.2 Conclusion

In conclusion, our thorough benchmarking has revealed that linked lists outperform both iterative and indexed approaches across the four topological queries, including *enters, leaves, crosses,* and *start and end search,* for both small and somewhat medium-sized datasets. As we observe the rate of increase to be faster for the non-linked list techniques, it indicates that linked lists could outperform them for larger datasets as well. Consequently, we are confident that linked lists have the potential to improve topological queries and overcome the limitations of the conventional approaches in use today. Hence, we can confidently answer the research question with yes, linked lists can be utilized for improving topological query performance. As we have seen in this project this method has a lot of potential, although only theoretical thus far. A lot of further work is required to use this method efficiently in a DBMS, but hopefully, this thesis shows that further research is worth pursuing.

Bibliography

[1] McKinsey Global Institute, "The internet of things: Mapping the value beyond the hype - executive summary," June 2015.

[2] D. Curry, "Strava Revenue and Usage Statistics (2022)." Available online at `https://www.businessofapps.com/data/strava-statistics/`; Last accessed: 06-12-2022, 2022.

[3] D. Pfoser, C. Jensen, Y. Theodoridis, and P. Greece, "Novel approaches to the indexing of moving object trajectories," 10 2001.

[4] H. Samet, *The Design and Analysis of Spatial Data Structures*. USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[5] P. Rigaux, M. Scholl, and A. Voisard, "6 - spatial access methods," in *Spatial Databases* (P. Rigaux, M. Scholl, and A. Voisard, eds.), The Morgan Kaufmann Series in Data Management Systems, pp. 201–266, San Francisco: Morgan Kaufmann, 2002.

[6] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, p. 47–57, jun 1984.

[7] A. N. Papadopoulos and Y. Manolopoulos, *The R-Tree and Variations*, pp. 13–21. Boston, MA: Springer US, 2005.

[8] A. Bhattacharya, *Fundamentals of database indexing and searching.* CRC Press, 2014.

[9] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*. Pearson, 7 ed., 2020.

[10] Y. Manolopoulos, A. Nanopoulos, A. Papadopoulos, and Y. Theodoridis, *R-Trees: Theory and Applications*. 01 2005.

[11] Oracle, "8.3.1 how mysql uses indexes." Available online at https://dev.mysql.com/doc/refman/8.0/en/mysql-indexes.html; Last accessed: 10-03-2023, 2023.

[12] P. Longley, M. F. Goodchild, D. J. Maguire, and D. Rhind, *Geographic Information Science and Systems*. Wiley Custom Learning Solutions, 4 ed., 2016.

[13] H. Lee, J. Romero, *et al.*, "Climate change 2023: Synthesis report," *IPCC*, 2023.

[14] Øyvind Bye Skille, "Fhi lager app for å spore folk i kampen mot koronaviruset," *NRK*, 2020. Available online at https://www.nrk.no/norge/fhi-lager-app-for-sporing-av-enkeltpersoners-bevegelser-i-kampen-mot-koronaviruset-1.14957299; Last accessed: 12-05-2023.

[15] N. Mu, Y. Wang, and P. Tian, "Spatio-temporal distribution characteristics of the cooperation between logistics industry and economy in southwest china," *International Journal of Computational Intelligence Systems*, vol. 15, no. 1, p. 7, 2022.

[16] B. Hunter, "How often does google timeline ping your phone's location?." Available online at https://dev.mysql.com/doc/refman/8.0/en/mysql-indexes.html; Last accessed: 24-04-2023, 2021.

[17] Esri, "History of GIS." Available online at https://www.esri.com/en-us/what-is-gis/history-of-gis; Last accessed: 06-12-2022.

[18] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini, and R. Montanari, "Big Spatial Data Management for the Internet of Things: A Survey," *Journal of Network and Systems Management*, vol. 28, pp. 990–1035, jul 27 2020.

[19] Open Geospatial Consortium, "Spatial Relations Defined." Available online at http://docs.safe.com/fme/html/FME_Desktop_Documentation/FME_Transformers/Transformers/spatialrelations.htm#DE9IM_Matrix; Last accessed: 01-12-2022.

[20] M. Vassilakopoulos and T. tzouramanis, *Quadtrees (and Family)*, pp. 2219–2225. Boston, MA: Springer US, 2009.

[21] K. Raptopoulou, M. Vassilakopoulos, and Y. Manolopoulos, "Towards quadtree-based moving objects databases," in *Advances in Databases and Information Systems* (A. Benczúr, J. Demetrovics, and G. Gottlob, eds.), (Berlin, Heidelberg), pp. 230–245, Springer Berlin Heidelberg, 2004.

[22] J Evans, "Personal heatmaps." Available online at https://medium.com/strava-engineering/personal-heatmaps-f51d15a0db2b; Last accessed: 01-05-2023, 2022.

[23] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pp. 322–331, 1990.

[24] S. de Konink, "R-tree.svg." Available online at `https://commons.wikimedia.org/wiki/File:R-tree.svg`; Last accessed: 21-02-2023, 2010.

[25] R. K. V. Kothuri, S. Ravada, and D. Abugov, "Quadtree and r-tree indexes in oracle spatial: A comparison using gis data," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, (New York, NY, USA), p. 546–557, Association for Computing Machinery, 2002.

[26] I. Kamel and C. Faloutsos, "Hilbert r-tree: An improved r-tree using fractals," tech. rep., 1993.

[27] D. Hilbert, "Ueber die stetige abbildung einer linie auf ein flächenstück," *Mathematische Annalen*, vol. 38, pp. 459–460, 1891.

[28] P. Morin, *Open data structures an introduction.* Athabasca University Press, 2014.

[29] M. T. Goodrich and D. M. Mount, *Data Structures and Algorithms in C++ 2nd Edition.* USA: Wiley, 2011.

[30] A. Crotty, V. Leis, and A. Pavlo, "Are you sure you want to use mmap in your database management system?," in *CIDR 2022, Conference on Innovative Data Systems Research*, 2022.

[31] Baeldung, "How to get the size of an object in java." Available online at `https://www.baeldung.com/java-size-of-object`; Last accessed: 21-02-2023, 05 2021.

[32] Y. Sun and M. Sarwat, "Riso-tree: An efficient and scalable index for spatial entities in graph database management systems," *ACM Trans. Spatial Algorithms Syst.*, vol. 7, jun 2021.

[33] M. Fourment and M. R. Gillings, "A comparison of common programming languages used in bioinformatics," *BMC Bioinformatics*, vol. 9, no. 1, p. 82, 2008.

Appendix A

Additional Figures

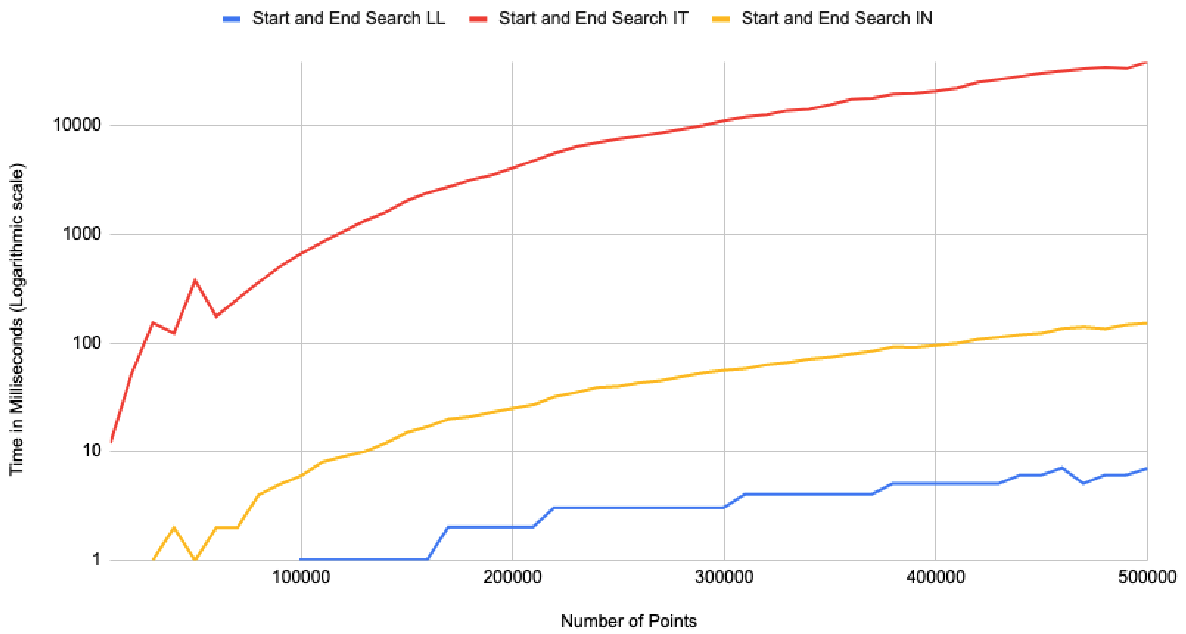


Figure A.1: Median time spent on each query for start and end queries with a logarithmic scale.

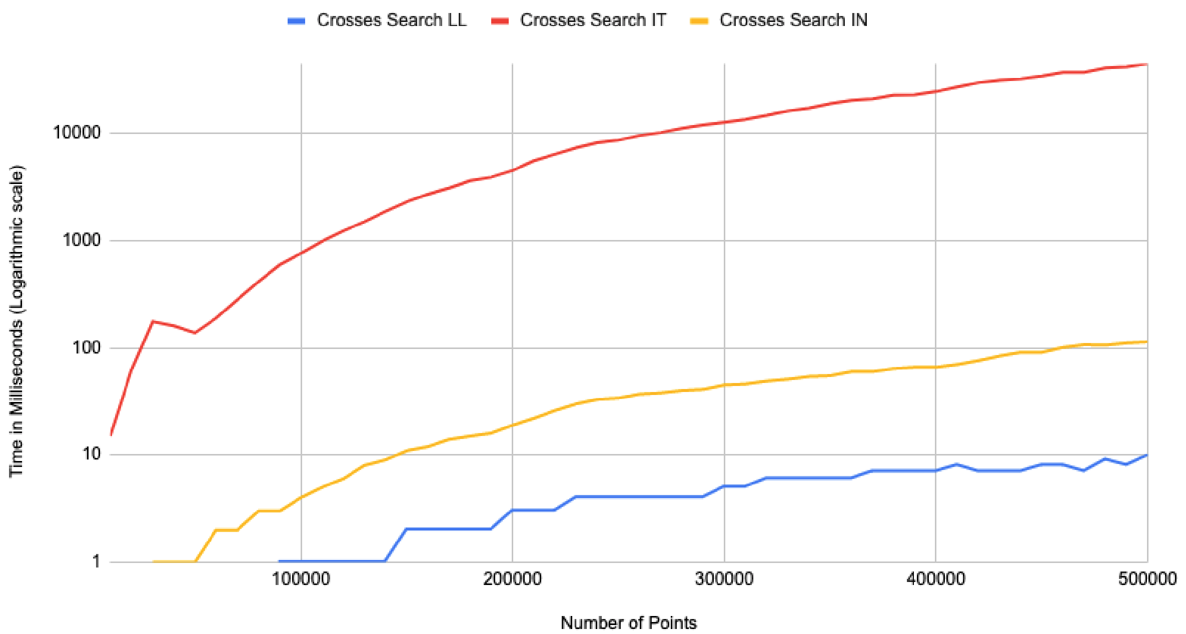


Figure A.2: Median time spent on each query for the crosses queries with a logarithmic scale.

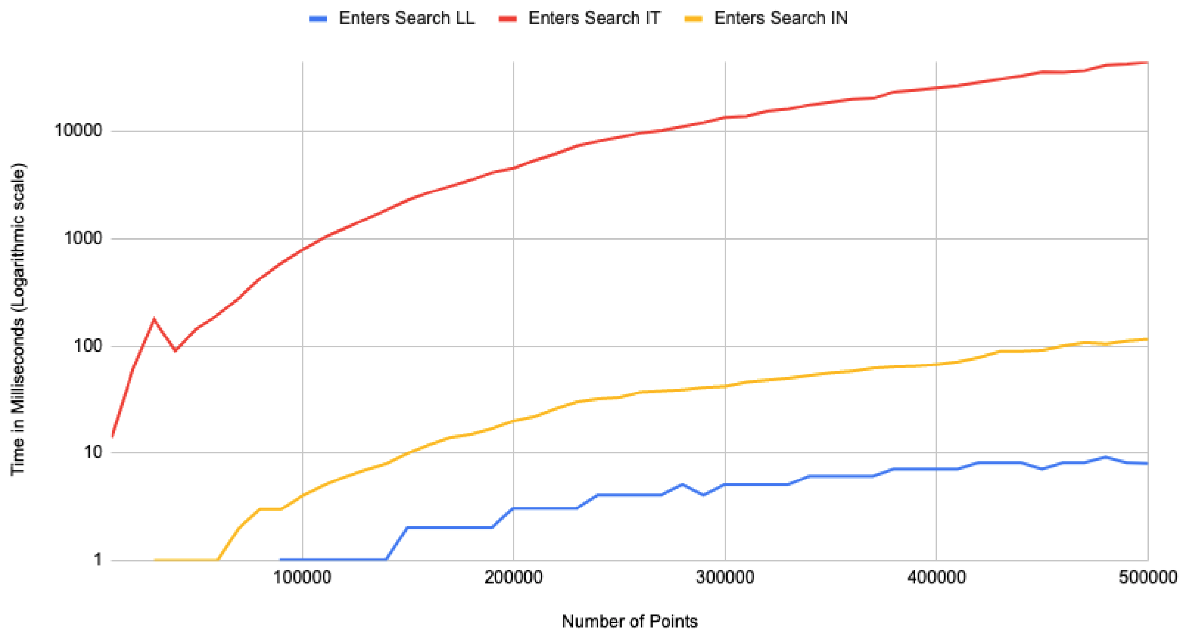


Figure A.3: Median time spent on each query for enters queries with a logarithmic scale.

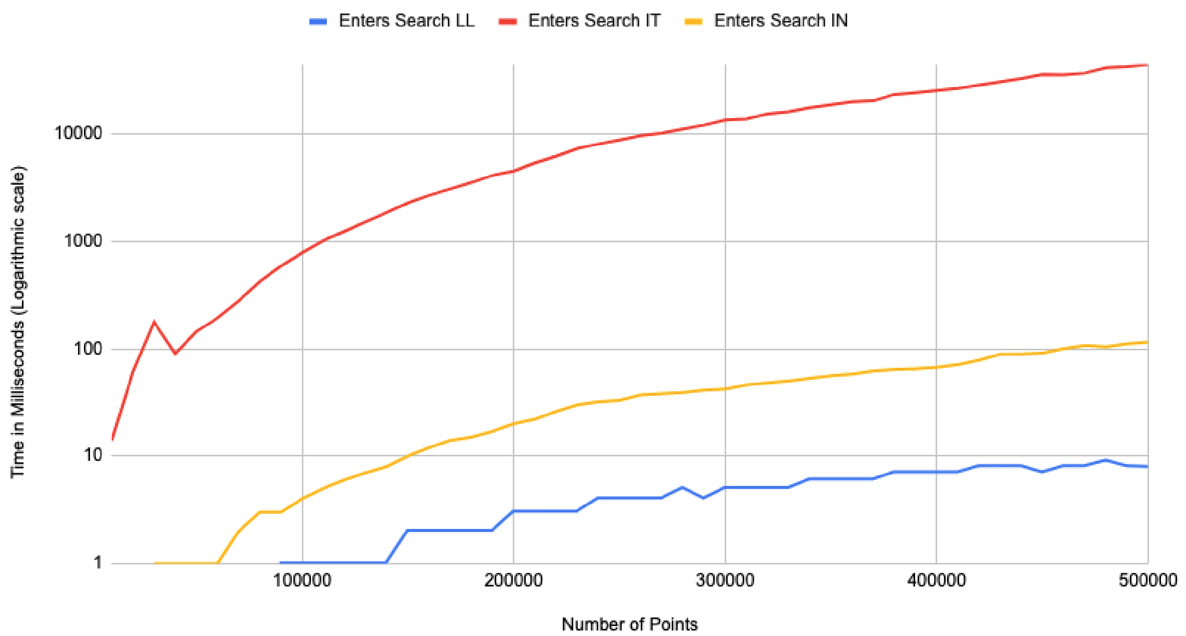


Figure A.4: Median time spent on each query for leaves queries with a logarithmic scale.

Appendix B

Scripts

```
import pandas as pd
import json
# Read data from CSV
df = pd.read_csv('train.csv')
# Remove all rows with missing data
df = df.drop(df[df.MISSING_DATA == True].index)
# Remove all unnecessary columns
df = df.drop(['MISSING_DATA', 'CALL_TYPE', 'ORIGIN_CALL', '
                                    ORIGIN_STAND', 'DAY_TYPE', '
                                    TIMESTAMP'], axis=1)

# Parse the string of coordinates
df.POLYLINE = df.POLYLINE.apply(json.loads)
# Create new rows with each pair of coordinates
df = df.explode('POLYLINE')
# Remove lines with missing coordinates
df = df[df['POLYLINE'].notnull()]
# Create new columns with the latitude and longitude split up
df[['LATITUDE', 'LONGITUDE']] = pd.DataFrame(df.POLYLINE.to_list(),
                                    index=df.index)
# Remove the old column with the lat,long pair
df = df.drop('POLYLINE', axis=1)
# Write to CSV
df.to_csv('converted.csv')
```

Code Listing B.1: Python script for manipulating the dataset and removing unnecessary data fields.

Appendix C

Data

Table C.1: Medians for each query. All times are given in milliseconds. An explanation for the abbreviations can be found in Table 1.1.

Points	SE-LL	SE-IT	SE-IN	C-LL	C-IT	C-IN	E-LL	E-IT	E-IN	L-LL	L-IT	L-IN
10000	0	12	0	0	15	0	0	14	0	0	13	0
20000	0	52	0	0	62	0	0	60	0	0	58	0
30000	0	153	1	0	176	1	0	177	1	0	178	1
40000	0	123	2	0	160	1	0	90	1	0	83	0
50000	0	374	1	0	138	1	0	144	1	0	143	1
60000	0	175	2	0	189	2	0	193	1	0	188	1
70000	0	254	2	0	282	2	0	278	2	0	283	2
80000	0	360	4	0	416	3	0	423	3	0	420	3
90000	0	504	5	1	594	3	1	585	3	1	576	3
100000	1	660	6	1	761	4	1	778	4	1	759	4
110000	1	845	8	1	985	5	1	1010	5	1	1008	5
120000	1	1052	9	1	1234	6	1	1239	6	1	1232	6
130000	1	1325	10	1	1505	8	1	1514	7	1	1551	7
140000	1	1594	12	1	1878	9	1	1847	8	1	1912	9
150000	1	2022	15	2	2303	11	2	2278	10	2	2287	10
160000	1	2405	17	2	2707	12	2	2680	12	2	2679	12
170000	2	2746	20	2	3104	14	2	3060	14	2	3172	13
180000	2	3140	21	2	3624	15	2	3510	15	2	3560	15
190000	2	3479	23	2	3916	16	2	4105	17	3	4046	17

Continued on next page

Table C.1 – continued from previous page

Points	SE-LL	SE-IT	SE-IN	C-LL	C-IT	C-IN	E-LL	E-IT	E-IN	L-LL	L-IT	L-IN
200000	2	4045	25	3	4520	19	3	4496	20	3	4559	19
210000	2	4743	27	3	5566	22	3	5340	22	3	5597	22
220000	3	5571	32	3	6403	26	3	6158	26	3	6257	26
230000	3	6351	35	4	7341	30	3	7275	30	4	7263	29
240000	3	6913	39	4	8214	33	4	8044	32	4	8057	32
250000	3	7489	40	4	8664	34	4	8748	33	5	8799	35
260000	3	7994	43	4	9587	37	4	9597	37	4	9801	36
270000	3	8580	45	4	10214	38	4	10160	38	5	10169	38
280000	3	9276	49	4	11195	40	5	11104	39	5	11492	40
290000	3	9989	53	4	12000	41	4	12034	41	5	12320	42
300000	3	11083	56	5	12731	45	5	13380	42	5	12698	43
310000	4	11934	58	5	13539	46	5	13755	46	5	14043	49
320000	4	12582	63	6	14735	49	5	15289	48	5	15445	49
330000	4	13724	66	6	16279	51	5	16116	50	6	16258	49
340000	4	14156	71	6	17335	54	6	17506	53	6	17533	53
350000	4	15464	74	6	19033	55	6	18705	56	6	18402	56
360000	4	17388	79	6	20357	60	6	19926	58	6	19952	58
370000	4	17875	84	7	21120	60	6	20292	62	7	20782	60
380000	5	19382	92	7	22749	64	7	23183	64	7	22153	62
390000	5	19787	91	7	23043	66	7	24066	65	8	24557	66
400000	5	20628	95	7	24605	66	7	25163	67	8	24080	72
410000	5	22043	99	8	27096	70	7	26462	71	8	25975	71

Continued on next page

Table C.1 – continued from previous page

Points	SE-LL	SE-IT	SE-IN	C-LL	C-IT	C-IN	E-LL	E-IT	E-IN	L-LL	L-IT	L-IN
420000	5	24864	108	7	29886	76	8	28325	78	8	30128	77
430000	5	26604	113	7	31523	84	8	30447	89	8	31414	83
440000	6	28365	119	7	32449	91	8	32771	89	8	32566	90
450000	6	30274	123	8	34336	91	7	35719	91	8	34236	91
460000	7	31711	136	8	37360	101	8	35371	100	8	38336	95
470000	5	33176	140	7	37421	107	8	36717	107	7	38192	105
480000	6	34424	135	9	41001	106	9	41201	104	9	41211	104
490000	6	33427	147	8	42009	111	8	42229	111	9	43123	110
500000	7	38443	152	10	44886	114	8	44389	115	9	45911	112

Table C.2: Data from R-tree implementation.

Points	Time	Depth	Regions	Leaf Nodes	Points per Leaf Node
10000	81.89853	7	2827	1807	5.534034311
20000	109.725696	8	5546	3551	5.632216277
30000	43.343771	9	8300	5328	5.630630631
40000	101.09047	9	11050	7095	5.63777308
50000	64.116653	9	14037	8971	5.573514658
60000	100.784615	9	16850	10780	5.565862709
70000	84.396346	9	19608	12558	5.574136009
80000	137.814328	10	22476	14373	5.56599179
90000	179.535872	10	25225	16123	5.582087701
100000	130.151428	10	27966	17871	5.59565777
110000	288.66225	10	30864	19709	5.581206555
120000	227.888707	10	33691	21505	5.580097652
130000	200.825123	10	36517	23324	5.573658035
140000	221.3416	10	39326	25112	5.575023893
150000	193.660454	10	42083	26874	5.581603036
160000	291.633109	10	44801	28626	5.58932439
170000	287.561288	10	47580	30403	5.591553465
180000	369.477911	10	50348	32164	5.596318866
190000	251.250956	10	53101	33930	5.59976422
200000	416.517202	11	55795	35658	5.608839531
210000	313.928302	11	58518	37393	5.61602439
220000	303.729293	11	61274	39159	5.618120994
230000	459.296052	11	63936	40876	5.626773657
240000	493.955362	11	66657	42617	5.631555483
250000	458.031746	11	69344	44358	5.635961946
260000	513.890283	11	72023	46070	5.643585848
270000	383.687745	11	74876	47885	5.638508928
280000	614.832013	11	77629	49631	5.641635268
290000	616.964954	11	80353	51369	5.645428177
300000	548.911425	11	83086	53105	5.649185576
310000	588.037176	11	85847	54864	5.650335375
320000	642.549443	11	88702	56670	5.646726663
330000	468.703217	11	91471	58454	5.64546481
340000	517.222202	11	94322	60270	5.641280903

Continued on next page

Table C.2 – continued from previous page

Points	Time	Depth	Regions	Leaf Nodes	Points per Leaf Node
350000	601.022285	11	97106	62057	5.639976151
360000	528.475469	11	99929	63828	5.640157924
370000	584.263606	11	102676	65586	5.64144787
380000	629.133591	11	105448	67356	5.641665182
390000	655.959803	11	108189	69127	5.641789749
400000	684.399556	11	110965	70914	5.640635136
410000	674.439799	11	113660	72644	5.643962337
420000	857.665603	11	116470	74425	5.643265032
430000	689.957683	11	119189	76163	5.645786012
440000	768.716947	11	121853	77890	5.648992168
450000	734.741901	11	124473	79574	5.655113479
460000	780.622617	11	127198	81321	5.656595467
470000	773.581576	11	129995	83110	5.655155818
480000	780.468687	11	132737	84874	5.655442185
490000	732.066564	11	135462	86627	5.656435061
500000	723.619389	11	138205	88398	5.656236566

Table C.3: Range search response times in milliseconds for the different amounts of data points.

Points	Median	Max	Min
10000	0	0	0
20000	0	0	0
30000	0	0	0
40000	0	0	0
50000	0	0	0
60000	0	0	0
70000	0	0	0
80000	0	0	0
90000	0	1	0
100000	0	1	0
110000	0	1	0
120000	0	1	0
130000	0	0	0
140000	0	1	0

Continued on next page

Table C.3 – continued from previous page

Points	Median	Max	Min
150000	0	1	0
160000	0.5	2	0
170000	0	1	0
180000	0	1	0
190000	1	2	1
200000	1	1	1
210000	1	2	1
220000	1	2	1
230000	1	2	1
240000	2	2	1
250000	1	1	1
260000	2	3	2
270000	1	2	1
280000	2	3	2
290000	2	3	2
300000	1	2	1
310000	1	3	1
320000	2	3	2
330000	2	3	2
340000	2	3	2
350000	2	2	2
360000	2	2	2
370000	3	4	3
380000	2	2	2
390000	3	4	2
400000	2	3	2
410000	3	3	2
420000	2	3	2
430000	2	3	2
440000	4	4	4
450000	4	4	3
460000	4	4	3
470000	4	4	4
480000	4	4	4
490000	4	4	4

Continued on next page

Table C.3 – continued from previous page

Points	Median	Max	Min
500000	4	5	4

Norwegian University of
Science and Technology