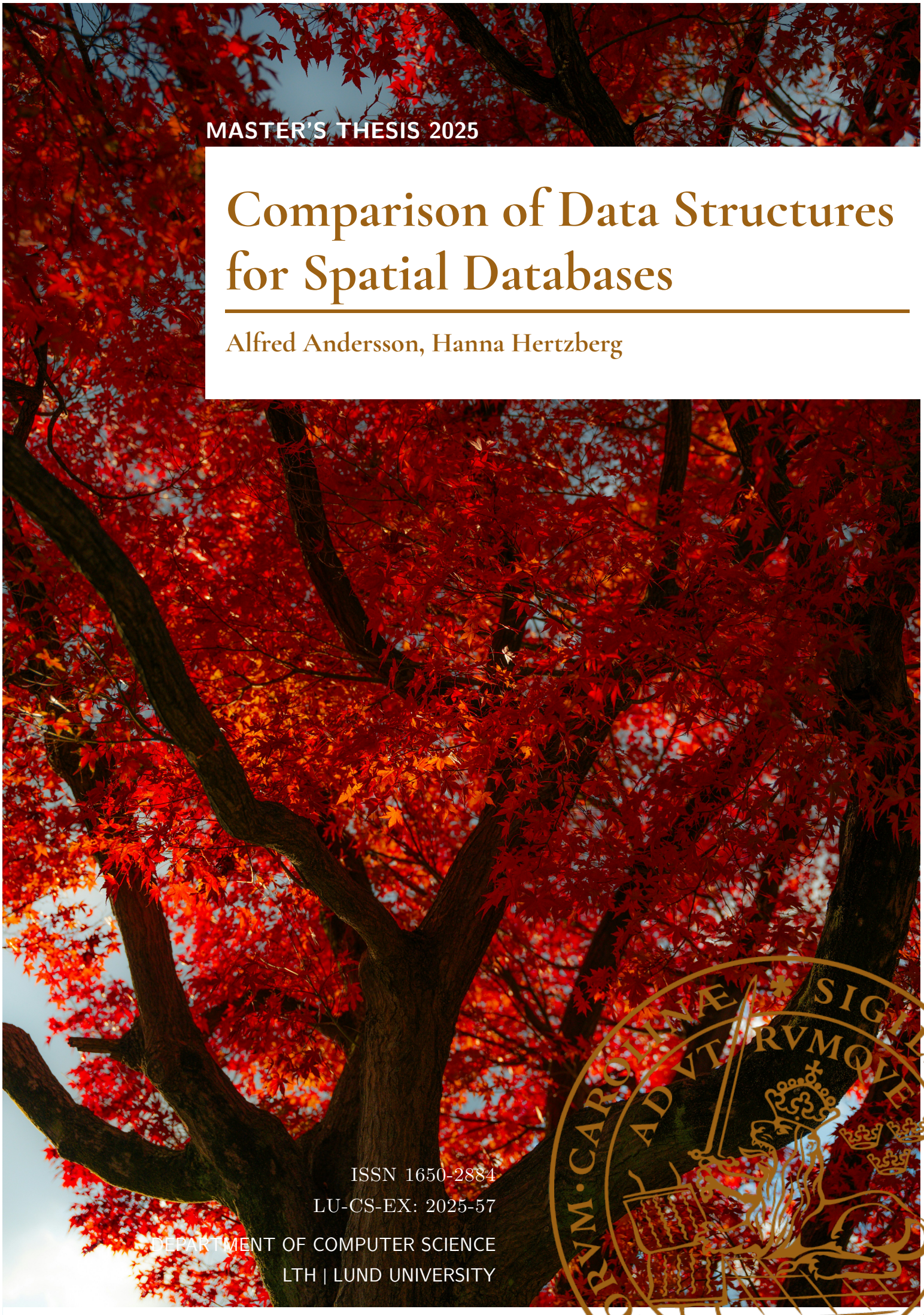


MASTER'S THESIS 2025

Comparison of Data Structures for Spatial Databases

Alfred Andersson, Hanna Hertzberg

ISSN 1650-2884
LU-CS-EX: 2025-57

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2025-57

Comparison of Data Structures for Spatial Databases

Jämförelse mellan datastrukturer för spatiella databaser

Alfred Andersson, Hanna Hertzberg

Comparison of Data Structures for Spatial Databases

Alfred Andersson
al8652an-s@student.lu.se

Hanna Hertzberg
ha1823he-s@student.lu.se

December 19, 2025

Master's thesis work carried out at a company in the in-flight entertainment business.

Supervisor: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

A spatial index is a data structure that accelerates queries on data that is inherently spatial. This has many applications—the specific use case we had in mind was maps used on in-flight entertainment displays, but there are also applications in other domains such as game development and computer-aided design.

This thesis compares the performance of three types of trees for use in such spatial indices—the point-region quadtree, the Guttman R-tree and the Hilbert packed R-tree. The goal was to find the fastest data structure for querying for points within a radius of a given point, and for, given a set of polygons, find the polygon(s) that a given point is inside of.

For the first problem, querying circles with a radius of 100 km on random geographic data, R-trees took up to around $1.80\times$ longer than quadtrees, depending on the size of the dataset. Hilbert trees and quadtrees were fairly close, with Hilbert trees taking between $0.95\times$ and $1.09\times$ as long as quadtrees, again depending on the dataset size.

In terms of performance, quadtrees and Hilbert trees were relatively close in most of our benchmarks but the R-tree was significantly slower than either. The R-tree was found to have poor structure, i.e. queries require searching a larger part of the tree than the other trees we tested. This was especially evident when using large numbers of points, even when querying very small areas.

The polygon problem had similar results, with the R-tree taking $1.08\times$ longer for one particular dataset.

Evaluation was done by measuring the execution time of queries on different datasets while varying parameters of the search trees. The code was also profiled using `operf`. All data structures and benchmarking code were written as a single-threaded C++ program, and the measurements were performed on an IBM Power AC922 server with 512 GB of RAM.

Keywords: C++, Hilbert packed R-tree, MSc, Quadtree, R-tree, Spatial index

Contents

Chapter 1
Introduction

When traveling with long distance flights, one could easily get bored [1]. Luckily most flights today have some kind of in-flight entertainment like movies and games, and even when that is not an option they usually have a map showing the plane's location. Unfortunately, these maps have a tendency to glitch and lag. This could be an annoying experience and could make the flight unpleasant. In this thesis we have investigated a way to improve this lag problem.

In a map application like an in-flight map, all cities, landmarks and time zones are stored in a spatial database. A spatial database is, as the name suggests, a database storing spatial information such as geographic or geometric data [7]. To store the data in a spatial database, the position of each data point has to be represented. This is usually done using the latitude/-longitude system [4].

In order to make use of spatial information when querying, a database can be enhanced with a spatial index. This is used to speed up the searching process and to sort data that is multi-dimensional. The spatial index is usually some kind of search tree, and the most common type of search tree is the R-tree structure in different variations [7].

This thesis was carried out at a company in the in-flight entertainment business.

1.1 Problem statement

In this thesis we aim to compare a few different data structures for use in spatial databases. They will be compared based on their performance on the following two tasks:

- Provided a set of points (locations on the Earth's surface), find all points within a given radius of a given point.
- Provided a set of polygons (representing time zones), find which polygon that a given point is inside of. See figure 1.1 for an overview of what this data looks like.

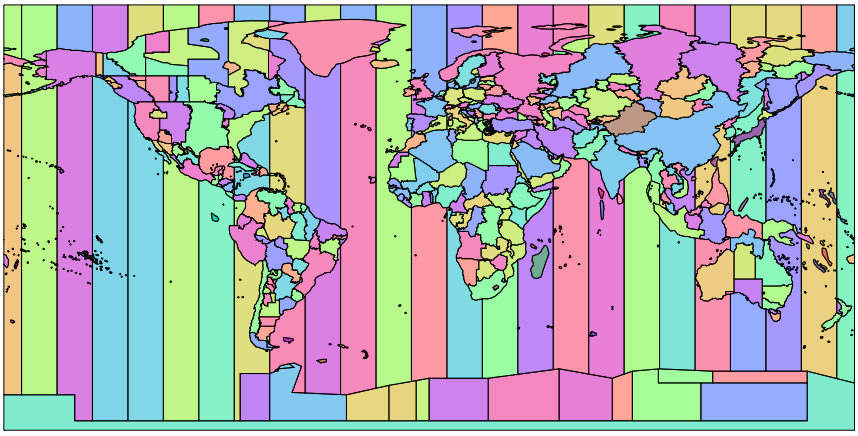


Figure 1.1: A visualization of the time zone dataset provided by the host company.

The given points represent locations on the surface of the Earth using latitude and longitude coordinates.

The data structures that were compared are quadtree, R-tree and Hilbert tree. These were chosen as they are some of the most commonly used data structures for spatial indexes, and are hopefully somewhat representative of the types of trees that would be considered for real use cases.

1.2 Research questions

The research questions for this projects are:

- What are the advantages and disadvantages of each data structure, considering memory usage and query execution time, for each of the above tasks?
- Is one of the data structures superior in general or does it depend on the situation? In particular, how does the size of the input data affect this?

1.3 Contribution

The primary use case we are targeting with this project is a map application in an in-flight entertainment system, which is where the concrete tasks we used to compare the data structures are from. The idea is to help our host company choose the most appropriate data structure for their use case.

Although several other authors have compared data structures for spatial indexes, e.g. [8][9][10], we could not find anything where the specific point-in-radius queries that we used were compared. We consider this to be our primary contribution.

Furthermore, some authors evaluated the trees for very specific use cases that were different from ours, e.g. [11].

Our results could also potentially be useful for other GIS (Geographic Information Systems) applications in which fast lookups of points in a specific area are needed.

1.4 Division of work

The work was divided evenly to maximize efficiency and learning.

Although both students participated in the literature search, in the initial stages Hanna did a larger portion of it.

The bulk of the implementation of the data structures was done by pair programming to ensure that both students understood all parts of the project. After they were mostly working Alfred did most of the bug fixing and ensured they shared a common API.

The benchmarking code was written by both students, where Alfred coded for both the radius and the polygon problem and Hanna only for the radius problem. Alfred also coded the memory usage tests. The actual running of the benchmarks was mostly done by Alfred, but we shared the data and evaluated it together. Hanna made most of the plots and tables.

Alfred made many visualizations of the data and of the different trees during the project to ensure everything was working properly, but these mostly didn't make it into the final report.

As for the report, Hanna generally focused on the first half, including introduction, theory, method and implementation, and Alfred on the second half focusing on results and discussion. However, both students contributed in every part of the report and the work was divided evenly and fairly.

Chapter 2
Background

This chapter presents some basic concepts and theory necessary to understand this thesis. We will start by briefly describing what spatial databases, spatial indexes and search trees are. Afterwards, we will talk about the specific type of searches we are interested in, and how that can be achieved using a spatial index. Finally, we will present each type of tree that we tested for this report and explain how they work.

2.1 General terminology

To introduce the subject of spatial databases and how they work, the basic structures have to be covered.

First off, a ***database*** is a structure that stores information. There are different types of databases, where the most common one is a relational database. A relational database stores information in tables, where each table has a fixed set of columns that are often defined in advance using a schema, seen in figure 2.1. Each row typically contains a primary key , some data, and possibly foreign keys that link to rows in other tables. The ability to link to other tables is what makes the database "relational" [3].

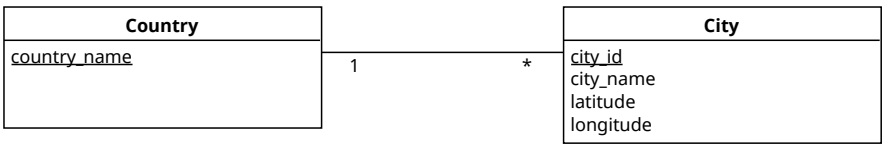


Figure 2.1: Relational database presenting the relationship between cities and countries. The underlined id:s are primary keys.

This thesis, however, examines spatial databases. ***Spatial database*** is the general term for a database that stores ***spatial data***, which could, for example, be a set of cities that each have a

latitude and longitude. This can be done in many ways, the most common being to enhance a relational database with a spatial index [7].

A ***spatial index*** is a data structure that accelerates various search operations on spatial data, which usually is some kind of search tree. The leaf nodes, explained further down in this section, in the search tree hold keys to a relational database where information is stored [7].

A ***search tree*** is a data structure used to store information of any kind. Search trees exploit structure in the data to accelerate searches—often this structure is in the form of an ordering that is imposed on the data. A search tree consists of a hierarchy of nodes: root, internal and leaf nodes. The root node is the start of the tree where both insertion and searching starts. Different trees are built differently, however all follow the same general idea where they consist of a root node that splits into a number of children. Depending on the type of tree, the number of children varies. If a child node has its own children nodes, this node is called an internal node. A node that does not have any children, and usually stores information, is called the leaf node [12].

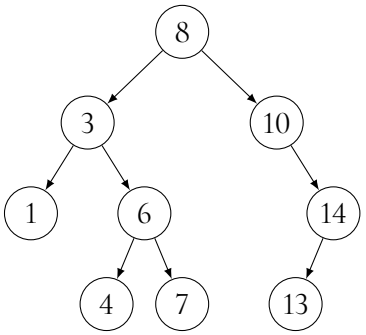


Figure 2.2: A binary search tree where the top node, containing number 8, is the root node, the nodes containing 3, 6, 10 and 14 are internal nodes and the rest are leaf nodes.

2.2 Theory

2.2.1 Search operations

All search trees we used are defined in terms of a two-dimensional coordinate system. The query rectangles (i.e. the inputs to search operations) are axis-aligned bounding boxes in this coordinate system. As the name implies, an AABB is a bounding box with axes aligned with the coordinate system's axes. In this case the axes are aligned with the x and y axes [17]. Both of the tasks we tested can be defined in terms of the same primitive search operation on a two-dimensional spatial index. This search operation finds and returns all entries (either points or rectangles) that intersect with a provided query rectangle.

For the first task, finding all points within a certain radius of another point, we first calculate an approximate bounding box for the circle's projection on the latitude/longitude plane. Then we calculate the actual distance from each returned point to the circle's center and use this to filter out all points that are not actually in the circle. The remaining points are then the result of the query.

For the second task, finding the polygon that a given point is within, we use a query rectangle with zero width and height to represent the given point. For each bounding box that this intersects with, we then perform a precise test to check whether the point is actually inside the polygon. Each point that passes this test is then returned.

In other words, both tasks can be performed by running the same search operation with an appropriate input, followed by filtering the output.

2.2.2 Coordinate systems

Since the Earth is not flat, there is no obvious "best" way to represent a point on its surface. Typically latitude and longitude are used as coordinates, but there are multiple ways to define these.

One of the most common coordinate systems, which is used by e.g. GPS, is based on the WGS 84 standard. This standard approximates the Earth's surface as an oblate spheroid with a radius of 6378.137 km along the equator and approximately 6356.752 km at the poles. After defining the prime meridian, the latitude φ and longitude λ coordinates can be defined in terms of this spheroid.

In our project, we chose to use a simpler but less accurate model to simplify the math. We approximate the Earth as a sphere with a fixed radius of 6371 km (a commonly accepted "nominal" radius of the Earth), which makes many calculations simpler. This will introduce some error in e.g. distance calculations, which could be a problem for certain real-world applications but should be insignificant for our goal of comparing the performance of different search trees. Using the "proper" model would likely make this part slower but we think it is unlikely that our results would change significantly.

Finding the distance between points

For our first task, after querying a spatial index, we need to filter out the points that are not actually inside the given circle. This is done by checking whether the distance from the center of the circle to each point is less than the radius of the circle. In this section we will explain how this can be achieved.

Let $R = 6371\,\text{km}$ be the mean radius of the Earth. We represent a point in coordinates, defined by:

$$\varphi \in [-\frac{\pi}{2}, \frac{\pi}{2}] \tag{2.1}$$

$$\lambda \in [-\pi, \pi] \tag{2.2}$$

λ wraps around such that λ is equivalent to $\lambda + 2\pi n$ for any integer n, but we canonically consider it to be in the range $[-\pi, \pi]$.

From this we can calculate the geocentric coordinates, i.e. 3D Cartesian coordinates with the center of the Earth as origin. Since we are only dealing with angles for now, we can assume that the Earth has radius 1:

$$\mathbf{x}_{\text{geocentric}} = \begin{pmatrix} \cos\varphi\cos\lambda \\ \cos\varphi\sin\lambda \\ \sin\varphi \end{pmatrix} \tag{2.3}$$

Given two geocentric points $\mathbf{a}$ and $\mathbf{b}$, the angle between them is given by:

$$\cos\theta = \mathbf{a} \cdot \mathbf{b} \tag{2.4}$$

Normally we would need to divide the right hand side by $\|\mathbf{a}\| \|\mathbf{b}\|$, but since both $\mathbf{a}$ and $\mathbf{b}$ have length 1 this is not needed.

Also, since we are on a sphere, θ is proportional to the distance between the points, since the circumference $2\pi R$ corresponds to 2π radians. Therefore, the distance between $\mathbf{a}$ and $\mathbf{b}$ along the Earth's surface can be found by:

$$d = R \arccos(\mathbf{a} \cdot \mathbf{b}) \tag{2.5}$$

Approximating a circle on the Earth's surface

To find all points within a radius of another point, we need to define a circle on the Earth's surface to search in. Although 2.5 from the previous section can be used for this, it is far too slow to simply calculate the distance to every point in our dataset. We therefore need a data structure that can eliminate most of the points that are sufficiently far away.

All of our data structures operate on points and axis-aligned bounding boxes, AABB, in Euclidean space. When querying for all points within an area, we need to provide a bounding box to search in. We will now present a way to calculate such a bounding box.

Consider a circle defined by its center point (φ, λ) and radius r, all in radians. We would like to find the width w and height h of a bounding box, i.e. a rectangle, that completely covers this circle.

First we need to define an equation for this circle. With $\Delta\varphi$ and $\Delta\lambda$ as coordinates relative to the circle's center, the equation would normally be:

$$\Delta\varphi^2 + \Delta\lambda^2 = r^2 \tag{2.6}$$

This would look like a circle when viewed on a flat map, but would not actually be a circle on the Earth's surface. This is because distances on the map do not correspond linearly to distances on Earth's surface.

To fix this, we need to introduce a scaling factor to $\Delta\lambda$, based on the distance each unit of λ corresponds to. We can derive this by calculating the length of the partial derivatives of $\mathbf{x}$:

$$\begin{aligned}
\left\| \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}\varphi} \right\| &= \left\| \begin{matrix} \frac{\mathrm{d}}{\mathrm{d}\varphi} \cos\varphi \cos\lambda \\ \frac{\mathrm{d}}{\mathrm{d}\varphi} \cos\varphi \sin\lambda \\ \frac{\mathrm{d}}{\mathrm{d}\varphi} \sin\varphi \end{matrix} \right\| = \left\| \begin{matrix} -\sin\varphi \cos\lambda \\ -\sin\varphi \sin\lambda \\ \cos\varphi \end{matrix} \right\| \\
&= \sqrt{\sin^2\varphi \cos^2\lambda + \sin^2\varphi \sin^2\lambda + \cos^2\varphi} \\
&= \sqrt{\sin^2\varphi + \cos^2\varphi} \\
&= 1
\end{aligned} \tag{2.7}$$

This result confirms that φ does not need a scaling factor, as changes to its value are always reflected uniformly on the sphere's surface.

$$\begin{aligned}\left\|\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}\lambda}\right\| &= \left\|\begin{matrix}\frac{\mathrm{d}}{\mathrm{d}\lambda}\cos\varphi\cos\lambda\\ \frac{\mathrm{d}}{\mathrm{d}\lambda}\cos\varphi\sin\lambda\\ \frac{\mathrm{d}}{\mathrm{d}\lambda}\sin\varphi\end{matrix}\right\| = \left\|\begin{matrix}-\cos\varphi\sin\lambda\\ \cos\varphi\cos\lambda\\ 0\end{matrix}\right\| \\ &= \sqrt{\cos^2\varphi\sin^2\lambda + \cos^2\varphi\cos^2\lambda} \\ &= \sqrt{\cos^2\varphi} \\ &= \cos\varphi\end{aligned} \tag{2.8}$$

The last step of 2.8 relies on the fact that the range of φ makes its cosine non-negative.

Since the length of this partial derivative is $\cos\varphi$, which does not depend on λ itself, we can use its reciprocal $\frac{1}{\cos\varphi}$ as a scaling factor to fix our circle. Since φ in equation 2.6 represents the center of the circle, we need to add the shift $\Delta\varphi$ as well, making the scaling factor $\frac{1}{\cos(\varphi+\Delta\varphi)}$ which leads to the equation:

$$\Delta\varphi^2 + \cos^2(\varphi+\Delta\varphi)\Delta\lambda^2 = r^2 \tag{2.9}$$

To find the height of the circle's bounding box, solve for $\Delta\varphi$:

$$\Delta\varphi = \pm\sqrt{r^2 - \cos^2(\varphi+\Delta\varphi)\Delta\lambda^2} \tag{2.10}$$

The maximum and minimum values for $\Delta\varphi$ occur when $\Delta\lambda$ is zero, which results in:

$$\Delta\varphi = \pm r \tag{2.11}$$

The height of the circle is therefore always $2r$, so we use $h = 2r$ for our bounding box.

To find the width of the circle's bounding box, solve for $\Delta\lambda$:

$$\Delta\lambda = \pm\frac{\sqrt{r^2 - \Delta\varphi^2}}{\cos(\varphi+\Delta\varphi)} \tag{2.12}$$

To find the maximum and minimum values of $\Delta\lambda$, we would need to differentiate the right hand side with respect to $\Delta\varphi$ and set it equal to zero. We were not able to find a closed-form solution to this, so we replaced the numerator with an upper bound:

$$|\Delta\lambda| \leq \frac{r}{\cos(\varphi+\Delta\varphi)} \tag{2.13}$$

Now, the expression will be larger the further $\varphi + \Delta\varphi$ is from zero, while assuming that $|\varphi + \Delta\varphi| < \frac{\pi}{2}$. Since φ is constant and we know the range of $\Delta\varphi$ from 2.11, we can determine the approximate range of $\Delta\lambda$, where we also use the fact that $\cos$ is an even function:

$$|\Delta\lambda| \leq \frac{r}{\cos(|\varphi| + r)} \tag{2.14}$$

Thus we can use the following formula for the width of our bounding box (the 2 comes from the fact that $\Delta\lambda$ can be both positive and negative):

$$w = \frac{2r}{\cos(|\varphi| + r)} \tag{2.15}$$

There is a special case when $|\varphi| + r \geq \frac{\pi}{2}$, which happens when the circle contains one of the poles. In this case, the denominator in 2.13 goes to zero and there is no maximum value for $\Delta\lambda$. In practice, we can simply use $w = 2\pi$ since that covers all possible longitudes.

2.2.3 Quadtree

A quadtree is in some ways similar to a binary search tree. In the binary search tree, every internal node is split into one or two new child nodes. For the quadtree every internal node is always split in four [5].

The four new children nodes each represent a quadrant of the map; north-west, north-east, south-west and south-east. When each child node is split, the quadrant is again divided to represent each direction, illustrated in figure 2.3. This is recursively done until a constraint is fulfilled and a leaf node is created containing the data points.

When searching for data points in the quadtree a search area, in the form of an AABB is used. The tree is made out of bounding boxes where each node stores a bounding box of the sub-quadrant's area. Because of this the searching process is quite easily done. Given the searching area, AABB, the search starts at the root and travels down the tree checking where the searching area overlaps with a node's bounding box. The search is done recursively so if an internal node's bounding box is overlapped by the search area, all children to the internal node will be visited [14].

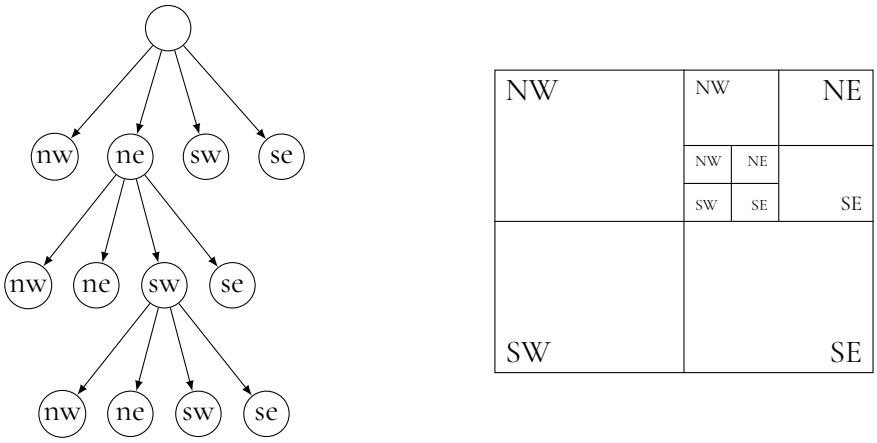


Figure 2.3: The quadtree structure and how the data is divided.

In the quadtree, all data points are stored in the leaf nodes. This is done to ease the searching process and to save memory [14]. The number of data points stored in the leaf nodes is called the M value. The value of M affects the composition of the tree a lot. A large M value results in a wide and shallow tree while a small M gives a narrow and deep tree.

There are different types of quadtrees, but for this project we chose to use the *point-region* (PR) quadtree. The point-region split is splitting all nodes into equal parts. When trying to insert a data point into a leaf node that already containts M points, the leaf node is split such that the node's bounding box is divided into four equal parts. Due to this, there could be leaf nodes that are empty, i.e. contain no points.

One of the largest limitations of the point-region quadtree is that the tree is unbalanced and does not consider the distribution of the points when splitting. Since it is unbalanced,

areas that are dense with data points will result in a very deep branch, while areas containing very few data points results in a shallow branch. Due to this, searching in the tree is suboptimal because data points situated close to each other might end up in very different branches of the tree. It also takes more time to search through a deep tree. This is, however, a problem all search trees struggle with, and tries to solve in different ways.

2.2.4 R-tree

The R-tree, short for rectangle tree, is a search tree that stores multi-dimensional data, and like the quadtree uses bounding boxes in the construction of the tree. A bounding box in the R-tree is the smallest box that could contain all data points in the node. Unlike the quadtree, the R-tree is balanced. A search tree is balanced when all leaf nodes are in the same layer, shown in figure 2.4. There are multiple ways to construct a balanced tree, and the procedure used for the R-tree is to expand on the width before the depth. When a leaf is full the node will be split in the same layer, instead of splitting into a new deeper layer [8].

The R-tree is also constructed in such a way that the bounding rectangles might intersect, which will lead to a slower search process. This is because when searching in the R-tree the search goes through every bounding box that is intersecting with the search area, where the search area again is an AABB. When bounding boxes are overlapping, the overlapping area will be searched multiple times. The search is the same for both points and polygons.

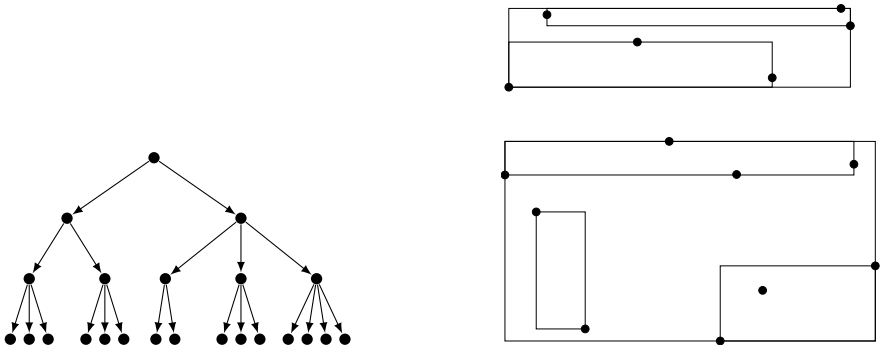


Figure 2.4: The right figure shows a set of data points and how they are grouped and the left figure shows a visual representation of the tree from the right.

A node in an R-tree does not have a fixed number of children like nodes in a quadtree do. Instead there is a maximum number of children a node can have, which we call the M value. Each node, except for the root, also has a threshold for the minimum number of data points that can be held, usually called m, which is equal to $\frac{M}{2}$. This is a consequence of the fact that nodes are split exactly in half when they are full, and are never split otherwise.

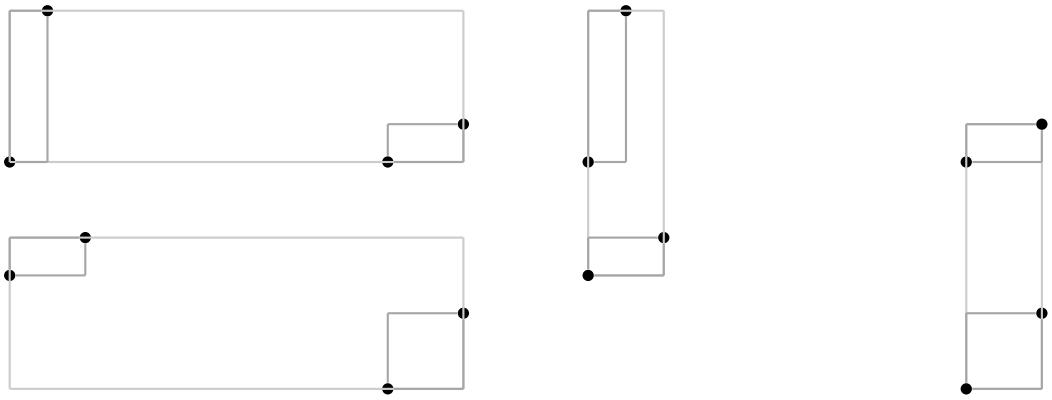
Although the M value for R-trees is defined differently to the M value for quadtrees in the previous section, it is still conceptually similar and we will later show that its effect on the performance characteristics of the tree is similar.

When a leaf node is full, i.e contains M data points, the leaf is split in two in the width, as mentioned earlier. After splitting, the tree is traversed from leaf to root, fixing the structure in case parent nodes might meet the threshold for number of children. If the root happens to

be split in this process, the tree will get one layer deeper. Otherwise the tree will only become wider.

One of the main downsides of the R-tree is the tendency of its internal nodes to overlap each other, forcing search operations to traverse a larger portion of the tree than would otherwise be necessary. However, depending on the way the R-tree is built, the amount of overlap varies.

This is called the "initialization", and there are a few different methods. For this project, we have chosen to use the original version of the R-tree, called the Guttman's quadratic split R-tree [8]. This initialization inserts one data point at a time, basing the insertion on the least enlarged bounding box area. The same criterion is used when splitting nodes, see figure 2.5.



(a) Bad split **(b)** Good split

Figure 2.5: A bad and a good split of bounding boxes containing points.

2.2.5 Hilbert packed R-tree

The Hilbert tree is a type of R-tree. There are two types of Hilbert R-trees, the dynamic Hilbert R-tree [10] and the Hilbert packed R-tree [9]. For this project we chose to use the Hilbert packed R-tree, both because the company we worked for use this type of Hilbert tree, and also since the implementation kept the R-tree's structure and only differed in the insertion process.

As previously mentioned, the structure of the Hilbert packed R-tree is the same as the R-tree, meaning that it is a balanced tree and is built by using bounding rectangles. A nice result of the Hilbert tree sharing structure with the R-tree is that the search is the same for both trees. The Hilbert tree is, unlike Guttman's quadratic split R-tree, constructed one level at a time from the bottom up by grouping nodes that are "close" to each other. For the grouping process the Hilbert curve is used.

The Hilbert curve is a space filling curve [9]. A space filling curve is basically projecting a multidimensional space onto a curve by getting arbitrarily close to every point in the space. For this project the two dimensional Hilbert curve was used, since the space we used, a map, is two dimensional. The Hilbert curve is constructed by a horse-shoe shape where every corner is representing a point in space. This is expanded depending on the desired accuracy, shown

in figure 2.6. From this curve we can derive a function that takes a 2D vector and outputs a scalar that we call the Hilbert value, which represents the input point's position on the Hilbert curve.

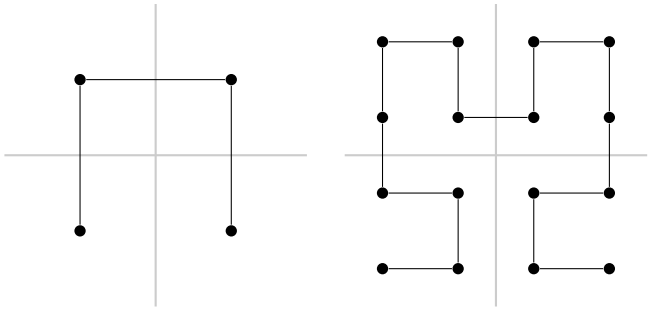


Figure 2.6: The Hilbert curve, first and second order. For the left picture the bottom left node is node 0, and the bottom right is 3. For the right picture, the bottom left is still node 0 and the bottom right is number 15.

Points being situated close to each other will most likely get a similar Hilbert number, however this may not always be the case. When looking at the right picture in figure 2.6, one can see that the bottom two points closest to the y axis, point number **2** and **14**, only have one unit's distance between them. However, on the Hilbert curve there are **12** points in-between. This is the main drawback of using the Hilbert curve.

As mentioned earlier, the Hilbert curve is used in the construction of the Hilbert packed R-tree. Before the data is inserted into the tree, it is sorted based on every point's Hilbert number. Since points close to each other most likely have similar Hilbert values, they will be put close to each other in the tree. All data is put into an array, and then, based on the M value, every chunk of M data points will each be put into a leaf node. When the array is empty, all nodes containing data are grouped into an internal node in chunks of M leaf nodes. This is repeated until there are M or less nodes left. The last internal nodes are put into the root node.

2.2.6 Definition of M value

Tree	Definition of M value
Quadtree	Maximum number of children of each leaf node. Internal nodes always have exactly four children.
R-tree	Maximum number of children of each node. In our implementation, the minimum number of children is $\frac{M}{2}$, except at the root.
Hilbert tree	Maximum number of children of each node. At most one node at each level of the tree can have fewer than M children.

Table 2.1: Definition of M value for each tree.

The M value has a slightly different meaning for each tree. See table 2.1 for an overview. As mentioned above, the overall impact of the M value is similar for all trees, which is why we still chose to call it the "M value" for all of them.

2.3 Related work

Quadtrees were originally proposed by [5], and the specific type of quadtree they presented is sometimes called the "point quadtree". It is very similar to a binary search tree in that each node, including internal nodes, is associated with a specific data point.

The quadtree we actually ended up using is the so-called point-region (PR) quadtree [18][13], which instead splits each node exactly in the middle and stores multiple points in each leaf node (and only in the leaf nodes).

The R-tree was introduced by [8], where the overall structure and two different initialization methods are presented. We used the "quadratic split" from this paper for our initialization, which results in a tree that can be called the "quadratic split R-tree" or simply the "Guttman R-tree". Our implementation of R-trees is based on pseudocode from this paper.

The R-tree has spawned many variants, including the R*-tree [2], which among other things has an improved heuristic for splitting nodes (i.e. where to put each point from the original node) which tries to minimize overlap. It also introduces the concept of "forced reinsertion" where portions of the tree are rebuilt dynamically to improve the overall structure. In the same paper, the R*-tree was shown to outperform Guttman's R-tree for many operations.

The Hilbert tree introduced by [10] is also an R-tree variant, which outperformed the R*-tree according to their measurements. We call this Hilbert tree the "dynamic" Hilbert tree, while the one we actually implemented is the Hilbert packed R-tree [9], which is built all att once from the bottom up. This Hilbert tree was compared against both Guttman's quadratic split R-tree and the R*-tree, and showed superior performance. No comparison was made to the dynamic Hilbert tree.

Another important work was [19] explaining the Hilbert curve and how to find the Hilbert number from the Hilbert curve. The article presents algorithms to both implement the Hilbert curve and how to calculate the Hilbert number from the Hilbert curve. The algorithms are general, such that they support an arbitrary number of dimensions.

[11] compared quadtrees and R-trees for a specific use case, the commercial "Oracle Spatial" database. The quadtree they use is a "linear" quadtree where the depth of the tree is uniform and chosen in advance. It appears that they use an in-house variant of R-trees, but not much detail is given as to exactly how it is implemented. For this use case, R-trees appear to consistently outperform quadtrees. However, given the specialized use case and lack of detail it is difficult to compare their results to those of other authors.

Chapter 3
Method

3.1 Approach

For this thesis we started by reading up on spatial indexes and various data structures that could be appropriate to evaluate. Popularity, ease of implementation, and relevance were some of the criteria we used when choosing which data structures to use. Based on this we ended up deciding to use quadtrees, R-trees and Hilbert trees.

After deciding on the data structures to use, we looked into libraries that implement them. While there were many libraries we could have used, we decided to implement everything ourselves. There were two main reasons for this—firstly, if we were going to use a library, we wanted to use the same library for all three data structures, to reduce the risk of the implementations being of drastically different quality. Secondly, implementing everything ourselves forced us to understand the data structures in depth from the beginning, reducing the risk of misunderstanding something and having to redo parts of our analysis later as a result.

The implementation of the trees for distance measuring purposes was carried out in three steps, starting with the quadtree, moving on to the R-tree, and ending with the Hilbert tree. Some limited testing was done continuously as we implemented each tree. Details about the implementation and the process is described in the next chapter.

Most of the code, including all data structures and benchmarking code, was written in C++, using only the standard library. We made use of templates for most of our data structures, with a generic `Data` parameter that determines what data is stored with each point[1]. Furthermore, all of our query functions (such as `find_in_radius`) take a generic parameter for a callback function, allowing custom behavior for each matched point without the overhead of a function call.

To verify that our implementations were correct, we ran numerous randomly generated

[1]For the polygon problem, this would store a pointer to a `Polygon` struct. Otherwise it was left blank, but still took up 8 bytes of space due to padding. In practice this would store use case specific data.

queries on each tree and compared the output to a brute-force approach, i.e. looping through every single point and checking whether each point is within the query region.

After the implementation and verifying that each tree returned the correct data, the testing scripts were written and tested. When testing the performance of the data structures we measured the runtime using the date and time module `chrono`, from the C++ standard library. As an additional validation step, each benchmark run printed the number of points found in total for each tree, in addition to the time taken. If these were not equal for each tree, then there had to be a bug in the implementations of at least one of them.

The benchmarking had to be divided into two cases for each of the tasks we tested. To test the structures for inside radius searching we found these three parameters useful to vary—the M value, the size of the search tree (i.e. the number of data points the tree contained) and the search radius.

The M value is an important parameter in the trees' construction, making it important to find a value that makes the trees comparable. To reduce the scope of the testing, we decided to compromise by finding a "good enough" M value for each tree that we then used throughout the rest of the benchmarks.

Both the number of points and the radius should have a significant impact on the execution time, so these were important parameters to investigate in our benchmarks. Our host company uses a dataset of approximately 40 000 points and a radius on the order of 100 km. We kept the radius for most tests where the radius was not the parameter being examined. As for the number of points, 40 000 is fairly small and we thought it would be more interesting to test with larger numbers of points, hence we decided to use a 10^6 as a baseline for many of our tests, and went all the way up to 10^9 when this was the primary parameter under consideration.

As input we mainly used randomly generated datasets of different sizes, but we also ran benchmarks using a dataset provided by our host company, and a random subset of nodes from the OpenStreetMap database [15]. The evaluation process is described later in this chapter.

For the OpenStreetMap data, we used the `osmium` [16] C++ library to enumerate all nodes in the OSM planet dataset, randomize their order, and output their coordinates to a file. When running a benchmark with n points, the first n points from this file were read and used as the input data.

When the structures were done being tested for the point in radius problem, they were modified to be able to handle polygons, considering the time zone problem. Since the quadtree is not compatible with polygons, i.e. can only take points as input, this was not used in the timezone problem. If the quadtree would have been modified, it would be too different from our original implementation and would not be comparable as a point region quadtree. The other search trees were polygon compatible and were hence used. This was followed by coding the benchmark for the time zone problem and later testing the code. For benchmarking we, again, used `chrono` from the C++ standard library, and we used a dataset of timezone polygons provided by our host company and a random dataset from OpenStreetMap.

3.2 Evaluation

3.2.1 Evaluation environment

All measurements were performed on an IBM Power AC922 server with the following specifications:

- Two POWER9 processors with 16 cores each and four hardware threads per core.
- 3.8 GHz clock frequency (verified using `cat /proc/cpuinfo`).
- 32 kB of L1 I-cache per core.
- 32 kB of L1 D-cache per core.
- 512 kB of L2 cache per pair of cores.
- 10 MB of L3 cache per pair of cores.
- 512 GB of RAM.

The server was running Ubuntu `22.04.5 LTS` with Linux kernel version `5.15.0-153-generic`.

The compiler used was `clang++`, which we compiled ourselves from source from commit `18edd41`, which is in between versions `21.1` and `22.0`. When compiling our program we used the flags `-O3 -flto`.

We used `cachegrind` to analyze the cache behavior of our program, using the following command: `valgrind --tool=cachegrind --I1=32768,8,128 --D1=32768,8,128 --LL=10485760,20,128 ./main`. Here we used `g++` version `15.1.0`, due to an issue we had with `clang` related to shared libraries.

3.2.2 Datasets

We used the following datasets for point data:

- Random points (geographical): These consist of latitude/longitude pairs that are uniformly distributed on the Earth's surface. This means that when displaying the points on a flat map, they look denser near the equator and sparser near the poles. This is the primary dataset we used.
- Random points (uniform): Uniformly distributed points in a rectangle. This was used when running queries without filtering with `is_inside_radius` in order to compare the results for our specific use case with a more "pure" benchmark that only measures the actual trees.
- Places dataset: Dataset provided by the host company containing around 40 000 points that represent cities or other points of interest. Since this dataset has a fixed size, it was skipped in any test where we varied the number of points. This was used both to check how generalizable our results were to a more realistic dataset, and because it is directly useful for our host company.

- OpenStreetMap: Random subsets of nodes from the OpenStreetMap database. Contains a total of approximately 10.1 billion points. A very large dataset that still has a realistic distribution of data. Like the places dataset, it was used to evaluate the generalizability of our results. Since it is so large, we were able to use subsets of different sizes to measure the impact of varying the dataset size.

All random numbers were generated using a pseudorandom number generator using fixed seeds, so that multiple runs using the same parameters will use the exact same set of random points.

For the polygon tests, we used the following datasets:

- Time zone polygon dataset: In general, this contains a polygon for each time zone/country pair. There are also some overlaps and the polygons have been simplified in a preprocessing step. Contains approximately 2000 polygons.[2] This was chosen because it is useful to our host company.

- OpenStreetMap: All polygons with the tag `boundary=administrative`. This includes country borders as well as all subdivisions inside each country. Due to including all subdivisions, most points will be in many polygons. This was chosen somewhat arbitrarily in order to compare it to the time zone dataset and ensure that the results do not only apply to that one.

3.2.3 Finding optimal M values

In order to ensure a fair comparison between the data structures, we first tried to find the optimal parameters for each of them. It is not feasible to test all possible combinations of M values and the other parameters used, hence we wanted to fix the M at a value that would work well for different kinds of data.

To find the optimal M for each tree, we varied M from 10 to 150, with step size 1, and measured the execution time of radius queries on three differently sized datasets of random points. The upper limit of the M interval was chosen based on the paper used for implementing the R-tree, since we thought the other search trees would have smaller M value due to their construction [8]. We also did some ad-hoc benchmarks with larger M values but did not see much impact on the performance, which is why we stuck to that range.

The M values were only tested on trees storing points and not polygons. The sizes of the datasets were chosen to be 10^4, 10^6 and 10^8 data points. This was because we wanted to test the M values on reasonably sized datasets but not use unnecessarily many datasets. We then plotted this and inspected the plots visually to find M values that perform well for every dataset. To differentiate signal from noise we ran each test twice with two different seeds.

The smaller datasets had 10^6 test cases while the large dataset had 10^4, in order to speed up the tests. One test case is one random point that symbolizes the plane's location, and is used as the center when searching within the radius.

The search used a radius of 100 km. This was because our host company used a similar radius in applications, and we wanted to make the test as similar to real life as possible.

[2] In reality, there are far less than 2000 time zones. Each "time zone" in this dataset is also paired with a region or city name, e.g. "Europe/Stockholm".

After deciding on M values for each data structure they were left fixed for the rest of the tests. There are no guarantees that this is optimal, and we leave verifying this to future work. We do not expect this to affect our conclusion significantly.

3.2.4 Radius problem

For these tests we randomly chose test points on the Earth's surface and measured the time taken to find all points in the dataset that were within a certain distance of that test point, i.e. a test case.

We then compared the execution time for each of our data structures while varying two parameters: the number of points in the dataset and the search radius. We ran separate tests for each dataset.

The random datasets are useful since they are easy to generate, but are not necessarily representative of real-world performance. The company dataset is actually used in production but is relatively small which limits the variety of tests we can run on it. To enable large-scale tests with realistic data we also included the OpenStreetMap dataset.

All of the random datasets are generated using a pseudorandom number generator using the same seed each time, ensuring that the results are repeatable.

To visualize our results, we saved everything to text files and then used the `matplotlib` Python library to plot the data neatly.

3.2.5 Polygon problem

For the polygon problem, there were not as many parameters to test. For example, there is no radius to vary since the input to each query is simply a point. Instead, we only tested the execution time for R-trees and Hilbert trees respectively, for each dataset.

The execution time was measured by generating 1 000 000 random test cases (just points in this scenario) and measuring the average time taken to find the polygon each random point was situated inside of. The random test cases were, as before, generated using a pseudorandom number generator using a fixed seed. 1 000 000 should be enough points that changing the seed will not significantly affect the measurements, but we did not actually verify this.

In these tests we tried two different datasets, one given by the hosting company containing around 2000 polygons, and the other one taken from OpenStreetMap.

3.2.6 Memory usage

To test the memory usage of each tree, we added methods to each tree to recursively visit each node and add up the size of all heap allocations in the tree, which mostly consisted of `vector`s, and in the case of quadtree also some `unique_ptr`s. We built each tree in the same way as in the "number of points" tests and simply called these methods to get the actual memory usage.

We only measured heap usage, not stack, as stack usage is very low and only proportional to the depth of each tree (i.e. $\log(N)$ in most cases).

For the `vector`s, the `size()` method was used rather than `capacity()`, meaning that any extra capacity in the vectors was not counted. This will slightly underestimate the

memory usage compared to what was actually used when running our benchmarks, but in a memory-constrained scenario you can simply reallocate the vectors by calling `shrink_to_fit()` to minimize the memory usage, matching our results.[3] If there is enough memory, having extra capacity is unlikely to affect performance in any meaningful way, since most of the spare capacity in the vectors will never be put in the CPU's cache anyway.

[3]Calling `shrink_to_fit()` is not free, and will make constructing the trees take longer. However, we are only concerned with the execution times of queries in this report, which are unaffected as they never allocate or modify heap memory.

Chapter 4
Implementation

4.1 Quadtree

When implementing the quadtree, we used C++'s `struct` to define the types of nodes and what they would contain.

We started by defining a generic node type. A generic node contains a bounding box and a variant. The bounding box matches the sub-quadrant of the map that the node represents, and the variant is used to store different data for internal and leaf nodes respectively. We used a package from the standard library in C++ called `std::variant` for this. The generic node is defined as follows:

```
struct Node {
    variant<LeafNode, InternalNode> node;
    BoundingBox boundary;
}
```

The generic node contains a bounding box which means that both the leaf node and internal node has a bounding box. The internal node holds unique pointers to the four directions the bounding boxes could be in: south-west, south-east, north-west and north-east. The leaf nodes instead holds all the data points in a `std::vector`. We didn't implement a special root node, instead we used an internal node as the root.

```
struct LeafNode {
    vector<Point> points;
};
```

```
struct InternalNode {
    Node nw;
    Node ne;
    Node sw;
    Node se;
};
```

The insertion process is straight forward. The data points are inserted one by one into the root node, first represented as a leaf node. When the root node is full, i.e the number of data points in the root is the same as the decided M value, and another data point is inserted, the node will be split into four new leaf nodes and the root node changed type to internal node. This process is then repeated but data points now are inserted into the leaf nodes and not into the root. This is done recursively until all data points are inserted.

4.2 R-tree

The R-tree was implemented similarly, with a default node containing a `std::variant` with the choices leaf node or internal node, however containing different attributes. Due to the implementation of the Guttman's R-tree, the default node had to contain a pointer to the parent node, but unlike the quadtree it did not contain the bounding box of the node [8].

```
struct Node {
    Node parent;
    variant<LeafNode, InternalNode> node;
}
```

The internal node in the R-tree, similar to the quadtree, holds references to its children. However, the R-tree's internal node used a `std::vector` holding internal entries, since there is no fixed number of children a node could hold. An internal entry symbolizes the child node, and is a pair of a (unique) pointer to a node, and the node's respective bounding box. A bounding box in the R-tree is the smallest box that could contain all data points in the node.

```
struct InternalNode {
    vector<InternalEntry> children;
};

struct InternalEntry {
    Node node;
    BoundingBox box;
};
```

Similarly, the leaf node in the R-tree contains an `std::vector` with leaf entries, where a leaf entry symbolizes the data points, and is made by data points' bounding boxes. For the first purpose of the thesis the data points were coordinates, so the bounding boxes were a middle point with height and width zero.

```
struct LeafNode {
    vector<LeafEntry> children;
};

struct LeafEntry {
    BoundingBox box;
};
```

The insertion process for the R-tree was similar to the quadtree's. It started with inserting one data point at a time into the root node, being represented as a leaf node at the start. This was continued until the root reached its maximum number of data points. When the root was full and another data point was to be inserted, the root was split into two leaf nodes. Then a new root node, now represented as an internal node, was constructed. The newly split leaf nodes were walked through and the new root node was added as their parent. This step was then repeated until another leaf node was full. Then the leaf was split in two new leaf nodes and the tree was walked from leaf to root rearranging the branch to be correct. If the root happened to be split the tree got one layer deeper.

4.3 Hilbert tree

The Hilbert packed R-tree had the exact same structure as the R-tree since the only difference between them was the insertion of data points in the tree. Therefore we kept the node structures and the leaf and internal node entries from the R-tree.

The insertion process, as explained in the theory chapter, was to insert all data points at the same time and group and regroup them until an R-tree was formed. The grouping was done by using Hilbert numbers derived from the Hilbert curve. Each data point was given a Hilbert number and based on these numbers the data points were sorted and then grouped ascendingly to form the tree.

To find the Hilbert number, we needed to find where on the Hilbert curve the data point was located. To be sure the data point would be on the Hilbert curve we used the Hilbert curve with the depth corresponding to the accuracy of the coordinates.

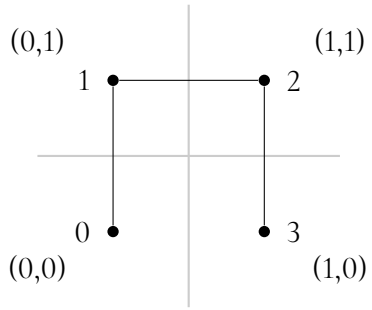


Figure 4.1: Mapping Hilbert numbers

As explained in the theory section, the first layer of the Hilbert curve covers four points in the four quadrants of a two dimensional space, and could be numbered 0 to 3 as in figure 4.1. When going deeper in layers, it can be seen that the first quarter of the Hilbert numbers is in

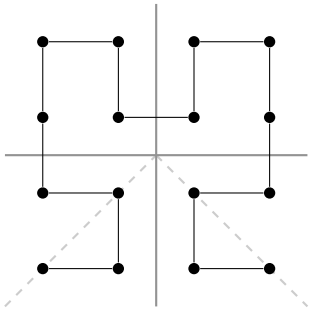


Figure 4.2: Second order Hilbert curve, could be seen as four first order Hilbert curves with the curves in the first and fourth quadrant being mirrored over different diagonals.

the 0,0 quadrant, second quarter in the 0,1 quadrant etc, where in the 0,0 and 1,0 quadrants the horse shoe shape is mirrored through the diagonal, see figure 4.2.

We used this phenomenon when computing the Hilbert number by first converting the coordinates of a data point to binary numbers. Then by numbering the quadrants as in figure 4.1 we knew that a 0 on the x axis was the left quadrants and a 1 on the y axis was the top quadrants. To then find the Hilbert number we took the the mapping in figure 4.1 and then converted the Hilbert numbers to binary numbers, as in table 4.1.

Quadrant	Hilbert number	Binary Hilbert number
0,0	0	00
0,1	1	01
1,1	2	10
1,0	3	11

Table 4.1: Corresponding Hilbert number to quadrant

Doing this conversion meant that the coordinates almost immediately gave the Hilbert number, by taking one bit from the left at a time and locating that on the space, see figure 4.3. The only change that had to be done was to mirror and invert the coordinates if they were in the bottom quadrants.

4.4 Polygon search tree

In the radius problem the search trees explained above contained data points, but for the time zone problem the data was time zone polygons. When storing data as points the quadtree stored the struct `Point` in the leaf nodes, while the R-tree and Hilbert tree stored the struct `BoundingBox`, with the center as a `Point` and with a height and width `0`. To store polygons we constructed a bounding box around the polygon, with the center point in the middle of the bounding box. This method only worked for the R-tree and Hilbert tree since quadtrees use points.

To make the R-tree and Hilbert tree compatible with polygons, we had to implement a help function such that when a polygon was to be inserted into the tree a bounding box would be constructed. We also constructed a new `is_inside_polygon()` function to check

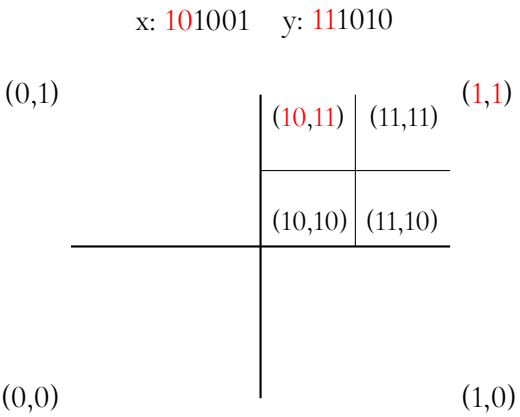


Figure 4.3: Illustration of how binary x and y coordinates work in a coordinate system.

if the given point was inside the found polygon or not. This was done by fixing the given point's y-coordinate and then going along the x-axis to the given point's x-coordinate counting the number of polygon-lines being crossed. If the number was odd the point was inside the polygon, if not the point was outside. We used a code snippet from [6] for the `is_inside_polygon()` function, which also handles various edge cases correctly.

Chapter 5
Results

5.1 Determining the best M values

As explained in the evaluation section the M value tests were performed on random datasets, containing 10^4, 10^6 and 10^8 data points respectively. In order to help differentiate signal from noise each test was run twice with two different seeds, and the results were then plotted in the same graph.

The search radius was 100 km and the small and medium sized datasets had 10^6 test cases while the large dataset had just 10^4 test cases in order to speed up the tests. Here a "test case" is simply a random point that defines the center of a circle, which is used as input to the search procedure.

Figure 5.1 shows the average execution time for quadtree queries on the small, medium and large datasets, for M values ranging from 10 to 150. There is no obvious best M value for the radius problem. For the small dataset the best M value seems to be around 35 to 40 and for the medium sized dataset the best M value is around 60 to 65. The large dataset has an inflection point at around M=45 but the performance keeps improving with larger M values.

While the plots for the small and medium tests clearly show an approximate minimum, the large test does not and therefore it's not obvious what M value to choose. We chose M=50 as a compromise for the quadtree since it performs well for the smaller datasets and for the large dataset the performance only improves very slightly above that value.

Figure 5.2 shows the results for R-trees in the same format as for quadtrees above. The plots are quite noisy which makes them harder to interpret, but some trends can still be observed. The small dataset works best with M values of around 40 to 80, while the medium and large dataset keep improving until around (very approximately) 50 and 80 respectively before flattening out.

Based on the plots we decided to use M=80. At this value the medium and large tests have both stopped improving, and it is just before the small test starts to perform worse again.

In figure 5.3, the results for the Hilbert tree are somewhat similar for the small and

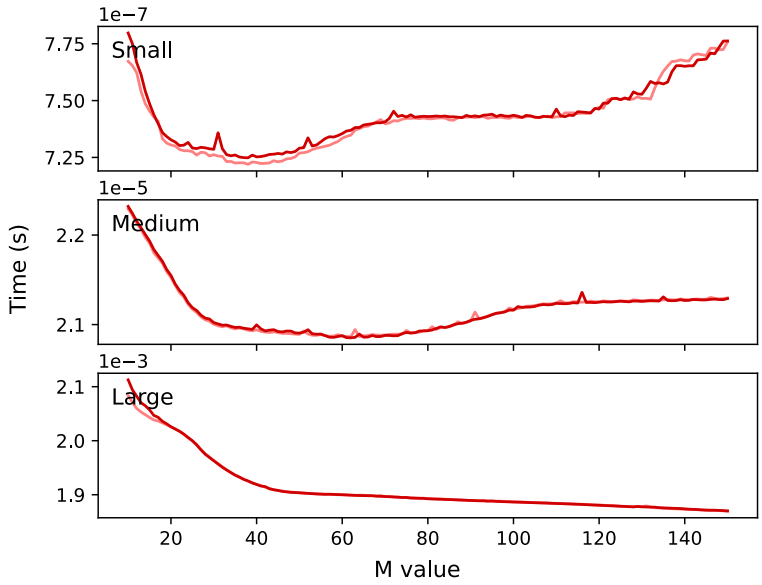


Figure 5.1: Execution time for quadtree queries with different M values on three sizes of dataset, measured in seconds. The queries return all points within 100 km of a given point. Each benchmark was run with two different seeds.

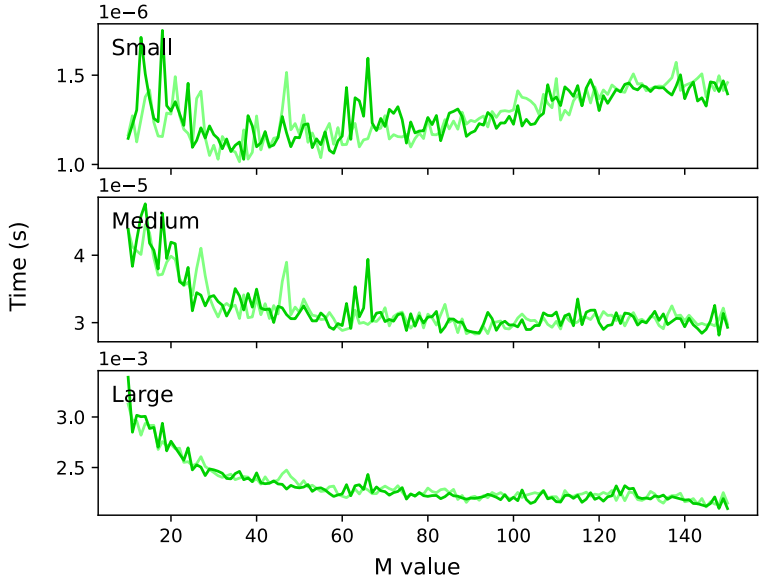


Figure 5.2: Execution time for R-tree queries with different M values on three sizes of dataset, measured in seconds. The queries return all points within 100 km of a given point. Each benchmark was run with two different seeds.

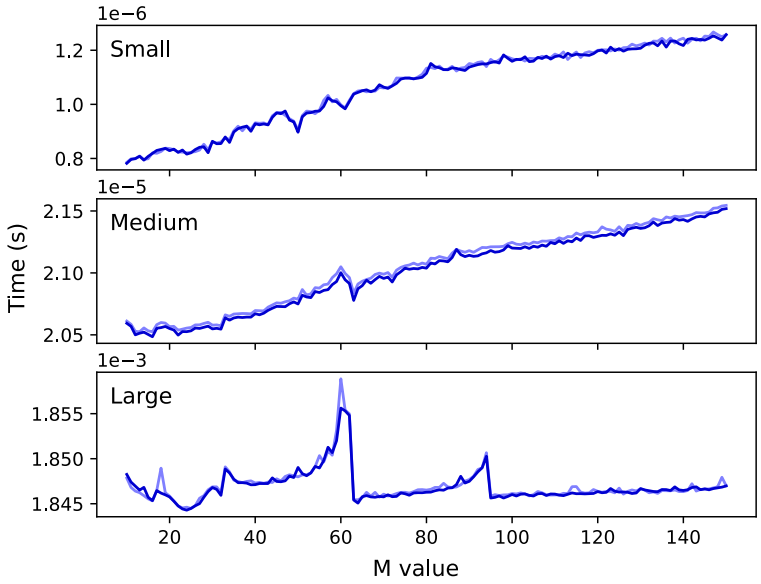


Figure 5.3: Execution time for Hilbert tree queries with different M values on three sizes of dataset, measured in seconds. The queries return all points within 100 km of a given point. Each benchmark was run with two different seeds.

medium datasets—small M values clearly give better results than larger ones.

For the large dataset the plot looks a bit irregular. It performs slightly better than average for M values around 10–30, 65–90, and 95–150. However, the range is quite small, with the maximum value only being approximately 0.5 % larger than the minimum. Thus the M value seems to not matter as much for as for the small and medium datasets.

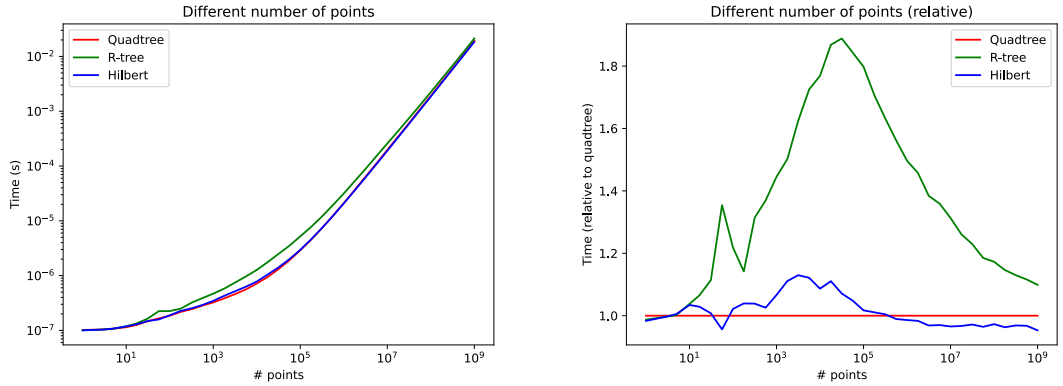
Since all Hilbert trees gave good results for lower M, we decided to use the M=10, the lowest value we tested.

5.2 Varying the dataset size

When testing the trees performances for different sizes of dataset (i.e. different numbers of points), we used 10^6 randomly generated test cases (points). The search radius used was 100 km for all trees and all datasets. We increased the number of points exponentially, plotting four values on the x axis for each order of magnitude. For each plot point, the size of the dataset is $10^{1/4}$ times greater than for the previous one. As found in the previous tests, the M values used were: **50** for the quadtree, **80** for the R-tree and **10** for the Hilbert tree.

As seen in figure 5.4 the time increases exponentially when searching for points in the increasing datasets. The trees start off with similar performance time, but quite quickly the R-tree performs the search much slower than the two other search trees. The Hilbert tree and quadtree perform similarly throughout all tests. As the dataset increases to 10^9 points all three trees perform similarly, however the R-tree is still the slowest.

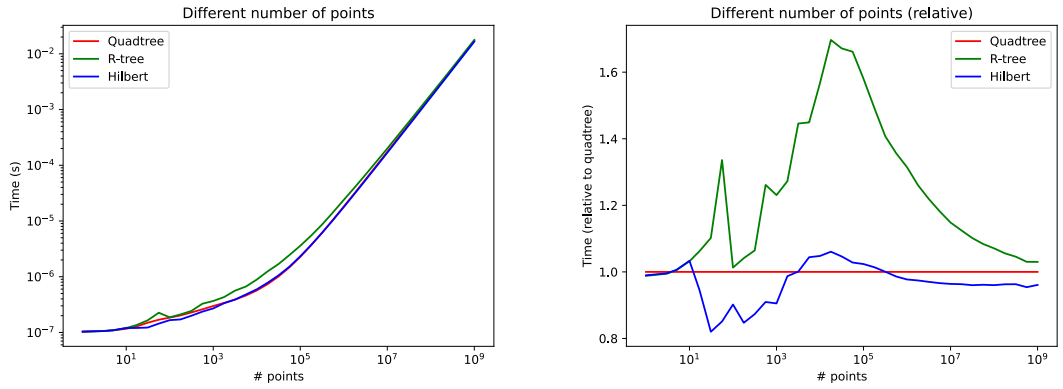
When looking at figure 5.5 (a) all search trees seem to perform similarly. However, when looking more closely, in subfigure (b), it can be seen that the R-tree over all performs the



(a) Absolute execution time

(b) Execution time relative to quadtree

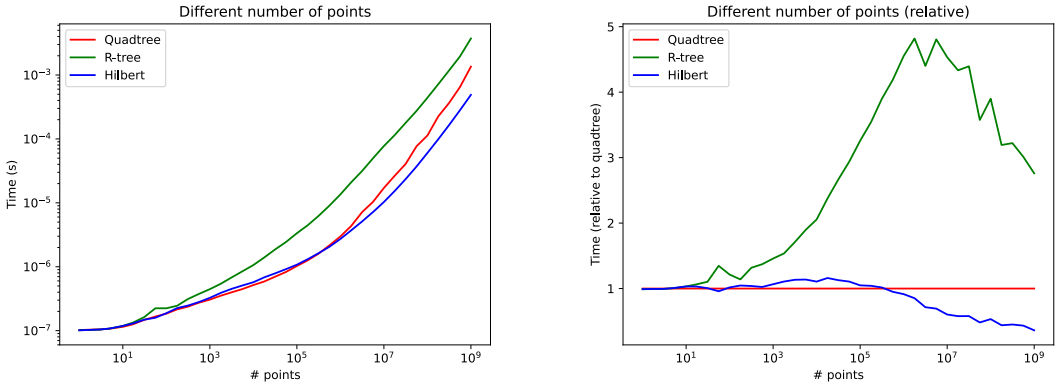
Figure 5.4: Execution time in seconds of queries on each tree for different numbers of random geographic data points.



(a) Absolute execution time

(b) Execution time relative to quadtree

Figure 5.5: Execution time in seconds of queries on each tree for different numbers of OpenStreetMap data points.



(a) Absolute execution time

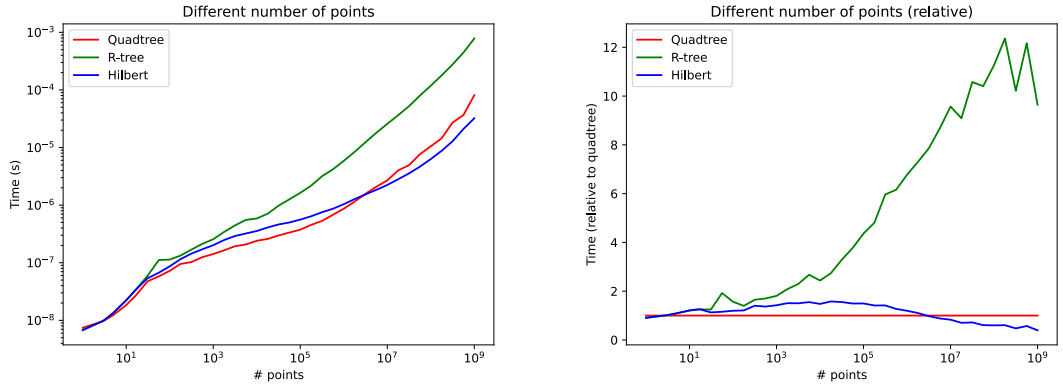
(b) Execution time relative to quadtree

Figure 5.6: Execution time in seconds of queries on each tree for different numbers of random geographic data points, excluding the `is_inside_radius()` function.

worst. For fewer datapoints the Hilbert tree performs the best, but as the number of points increases the Hilbert tree and the quadtree performs similarly. One interesting detail is that when the number of points increases, the R-tree's performance increases compared to the other trees. At the worst point a little over 10^4, the R-tree performs almost 70% worse than the quadtree while the Hilbert tree only performs around 5% worse. We do not believe that 10^4 is particularly special—most likely the two opposing effects (R-trees getting relatively worse and `is_inside_radius()` taking up a greater proportion of the execution time) simply happen to line up in a way that produces the peak in the relative plot. Although we did not verify this, the peak should move towards the left when increasing the query radius as the latter effect will be more significant with a larger radius that makes the query return more points.

Figure 5.6 shows some interesting results. The figure plots the search time for an increasing amount of data points, however without taking the function `is_inside_radius()` into account. This was to see if the function had a large impact on the search time, i.e. we only measured the time it took to find a bounding box with the wanted radius as the sides in the trees. It can be seen that all trees are faster and the difference between them is larger. At a very small number of points the trees perform kind of the same but as the number of points increases the R-tree takes more time than the other two trees. Around 10^7 data points the R-tree is almost 5 times slower than the quadtree, which can be seen in figure 5.6 (b). The figure also shows that the more data points, the more beneficial it is to use the Hilbert tree.

In figure 5.7 an increasing number of uniformly distributed data points is plotted. By "uniformly distributed" we mean uniformly in a rectangle, not on the Earth's surface. On this data we then ran queries on rectangular areas without any filtering, simply returning every point that is in the rectangles. Specifically, all random points were within a square with side length 1, and the query regions were squares with a side length of $\frac{1}{500}$. The idea was for the query areas, and thus also the number of points returned by each query, to be about the same for each dataset size as in the previous tests that used geographic data. However, due to a mistake in our calculations the area ended up being around four times smaller than



(a) Absolute execution time

(b) Execution time relative to quadtree

Figure 5.7: Execution time in seconds of queries on each tree for different numbers of random uniform data points.

intended.

As seen in figure 5.7 (b) the trees' execution times follow the same trend as in earlier tests. It becomes obvious that the Hilbert tree is superior for a larger number of points.

In figure 5.8 the plot shows the performance for an increasing dataset but where the radius decreases proportionally, i.e. the number of points inside the searching radius will be approximately the same for every dataset. Again, the quadtree and the Hilbert tree perform similarly throughout the test while the R-tree's search time increases drastically around 10^4 data points.

5.3 Varying the query radius

When testing how the size of the radius affects the execution time, the dataset contained 10^6 points and the M values chosen were, as before, 50 for the quadtree, 80 for the R-tree and 10 for the Hilbert tree.

Figure 5.9 plots the search time when changing the size of the radius for the different trees. As seen both the quadtree and the Hilbert tree are a lot faster than the R-tree for small radii, but when the radius increases, so does the search time. Around 10^4 the search time increases drastically for all trees, however the quadtree and the Hilbert tree increase a lot more. The R-tree is still slower but the difference between the trees is much smaller.

The plot in figure 5.9 also shows some interesting results where the plot stagnates around 10^7 meters. This is because the coverage area starts to reach the area of the Earth. This means that even as the radius increases all data points are covered.

5.4 Places dataset

The "places" dataset provided by the hosting company, consisting of around 40 000 points, was also used for a final point-in-radius search. The results, seen in table 5.1, were in line

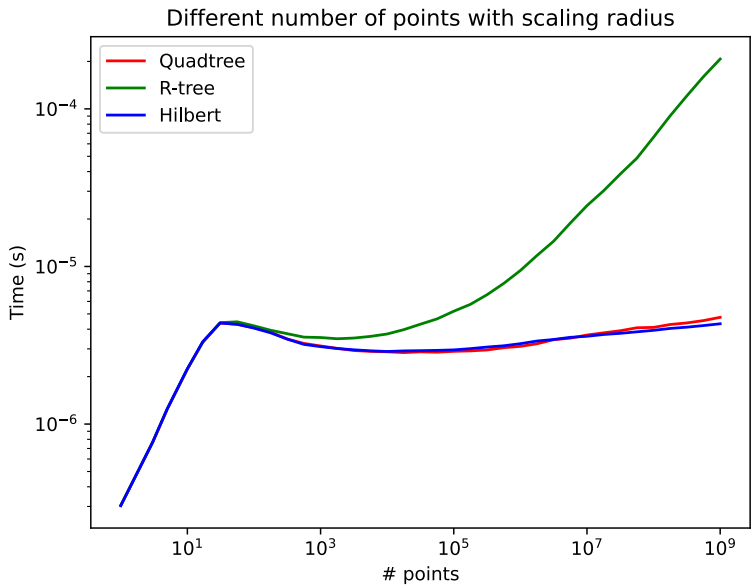


Figure 5.8: Execution time in seconds of queries on each tree for different numbers of random geographic data points. In this benchmark, the query radius was scaled so that each query returns approximately 6 points on average.

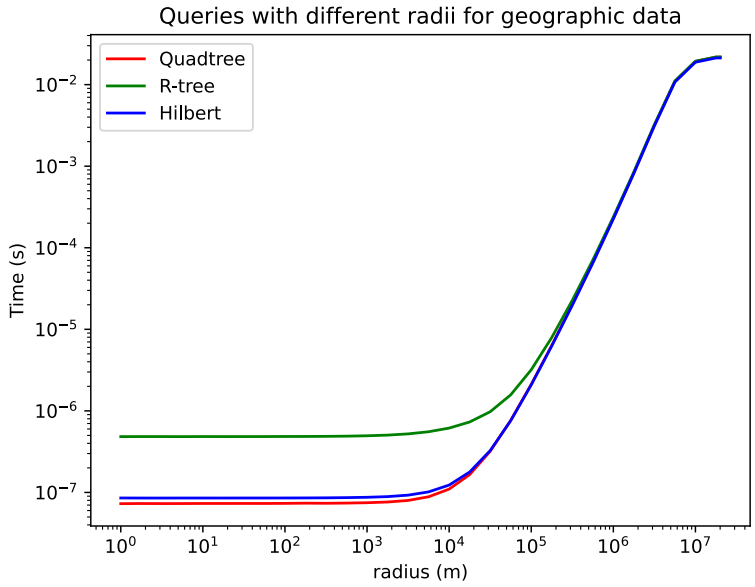


Figure 5.9: Execution time in seconds of queries with different radii for each tree on a set of 10^6 random geographic data points.

Quadtree	R-tree	Hilbert tree
1.0070	1.3675	1.0716

Table 5.1: The execution time in µs of queries for points within a 100 km radius of a given point, on the host company's dataset.

with the other tests—for "small" datasets of less than approximately 10^6 points the quadtree was usually faster than Hilbert trees, which is also the case here. As expected R-trees were the slowest.

5.5 Point-in-polygon evaluation

	R-tree	Hilbert tree
Time zones	1.424 µs	1.323 µs
OSM adm. boundaries	298.9 µs	296.9 µs

Table 5.2: Execution time for the point-in-polygon tests. The number of test cases run was 10^8 for the timezone dataset and 10^7 for the OSM dataset.

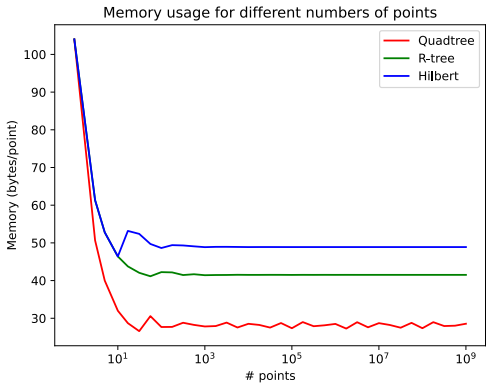
Table 5.2 shows the execution time for which polygon the wanted point is inside of. For the time zone dataset, Hilbert trees took around 7.1% less time than R-trees. For the OSM administrative boundaries, the difference was only 0.7%.

We verified that our implementations were correct in three separate ways—first, for a smaller sample of queries we verified that each tree returned the same result. We also, as with points in the radius tests, always printed the sum of the number of polygons returned for each of the 10^6 test cases, separately for each tree. This is extremely unlikely to end up being equal if the trees actually returned different results. Lastly, we generated an image based on figure 1.1, but with a large number of points plotted and colored based on which polygon they are inside of. This was then visually inspected to ensure that we could not find any points that were given the wrong color. For this visual test we only used the Hilbert tree, as it was mostly intended to test the `is_inside_polygon()` function—the consistency between the trees was verified in other ways as described above.

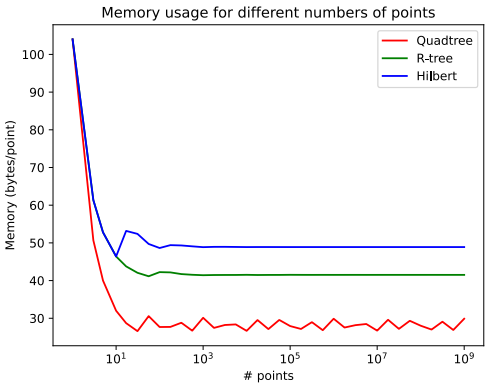
5.6 Memory usage

Figure 5.10 shows the memory usage for each tree in different situations. The memory usage is generally consistent between different types of data, especially for R-trees and Hilbert trees. The only discernible difference is the fact that the quadtree's memory usage is more consistent with less random data, which can be clearly seen in the graphs. For the Hilbert tree specifically, the memory usage will always be exactly the same for a particular number of points, due to the bottom-up approach used in its construction.

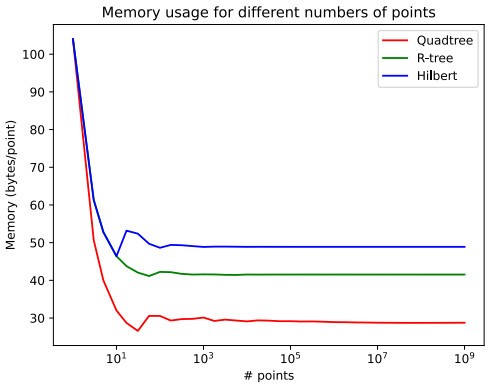
The reason quadtrees look so good in this data is that our implementation of them is *less* generic than the other trees. For R-trees and Hilbert trees we wanted to support both poly-



(a) Random geographic data.



(b) Random uniform data.



(c) OpenStreetMap data.

Figure 5.10: Memory usage per point for different trees, datasets, and numbers of points.

gons and points, so we made them store bounding boxes in the leaves. When inserting points into these trees they are therefore stored, as mentioned in the theory section, as bounding boxes with zero width and height, wasting 16 bytes per point. When correcting for this, Hilbert trees end up at ≈34 bytes/point and R-trees at ≈26 bytes/point, placing quadtrees right in the middle in terms of memory efficiency. This implies that the memory usage of each tree mostly depends on the M value, and not on the specific type of tree.

5.7 Profiling results

In order to find out what takes the most time in our benchmarks we profiled the program using `operf`. The `operf` tests were run with the `-O0` and `-fno-inline` flags to keep the assembly somewhat similar to our C++ code, and to preserve all function calls. The dataset used was 10^6 random nodes from the OpenStreetMap dataset, and 10^6 test cases were run for each tree. The `CYCLES` performance counter was used to estimate the number of CPU cycles spent in each function.

Figure 5.11 shows an overview of the results of these tests. We can see, for example, that trigonometric functions such as `sin`, `cos`, and `acos` take up a significant portion of the runtime for all three trees. These are only ever called from `is_inside_radius`, and `to_cartesian` which is also only called from `is_inside_radius`. Since `is_inside_radius` essentially acts as a filter on the output from the queries on the actual trees (which simply find all points withing a bounding box), the number of calls and the total time in these functions are approximately the same for each tree.

The first thing that you might notice in 5.11 is the large number of samples in `intersects` and `find_in_box` for the R-tree. This illustrates the reason for the R-tree's inefficiency—it simply has to visit many more nodes for each query because of its poor structure with many overlapping internal nodes.

The `opreport` results do not separate the process of constructing the trees, which is not measured in our other benchmarks, from the actual queries. The functions that are used when making queries (and take significant time) are the following:

- For all trees: `AABB::intersects`, `is_inside_radius`, `to_cartesian`, and trigonometric functions.
- For quadtrees: `quadtree::Node::find_in_box` and `AABB::contains`.
- For R-trees and Hilbert trees: `rtree::Node::find_in_box`.

Using `opannotate -a`, we also analyzed the assembly of the functions identified above, to find what specific parts of the code take the most time. `opannotate -s` (i.e. source-based annotations) could not be used due to an issue related to our use of C++ templates. This issue is described in more detail in section 6.6 in the discussion regarding implementation complexity.

In the quadtree test, `quadtree::Node::find_in_box` consists mostly of loads, stores, conditional branches and a few simple arithmetic and logical operations. Of the 37368 samples in this function, 8359 were in a `ld` instruction that loads points from the vector inside a leaf node in a loop. There were a few more load instructions that also took up significant

```
samples  %        image name   symbol name
153327   23.6072  libm.so.6    sinf32x
72129    11.1054  main         to_cartesian(Point)
61475     9.4651  libm.so.6    __acos_finite@GLIBC_2.17
57215     8.8092  main         AABB::contains(Point) const
45181     6.9563  libm.so.6    cosf32x
39306     6.0518  main         is_inside_radius(Point, Point, double)
37368     5.7534  main         quadtree::Node::find_in_box(...)
15069     2.3201  main         CartesianPoint::CartesianPoint(...)
14664     2.2578  main         std::pair<...>(...)
12528     1.9289  main         AABB::intersects(AABB) const
11805     1.8176  main         std::abs(double)
```

(a) Results for Quadtree.

```
samples  %        image name   symbol name
331711   16.2346  main         AABB::intersects(AABB) const
212432   10.3968  main         AABB::from_boxes(...)
158894    7.7766  main         rtree::Node::find_in_box(...)
154044    7.5392  libm.so.6    sinf32x
89823     4.3961  main         GuttmanRtree::split_node(...)
74032     3.6233  main         to_cartesian(Point)
64598     3.1616  main         Point::Point(double, double)
62109     3.0397  main         std::abs(double)
60912     2.9812  libm.so.6    __acos_finite@GLIBC_2.17
51494     2.5202  main         Node::calc_bbox()
43368     2.1225  libm.so.6    cosf32x
42640     2.0869  main         GuttmanRtree::choose_leaf(...)
39995     1.9574  main         is_inside_radius(Point, Point, double)
```

(b) Results for R-tree.

```
samples  %        image name   symbol name
151526   22.9725  libm.so.6    sinf32x
72413    10.9784  main         to_cartesian(Point)
68020    10.3123  main         AABB::intersects(AABB) const
62084     9.4124  libm.so.6    __acos_finite@GLIBC_2.17
44811     6.7937  libm.so.6    cosf32x
38921     5.9007  main         is_inside_radius(Point, Point, double)
36581     5.5460  main         rtree::Node::find_in_box(...)
14950     2.2665  main         CartesianPoint::CartesianPoint(...)
13270     2.0118  main         hilbert_int(unsigned int, unsigned int)
10327     1.5657  main         std::abs(double)
```

(c) Results for Hilbert tree.

Figure 5.11: `opreport` results for each type of tree, measuring clock cycles—each sample represents 100 000 clock cycles. Some of the symbol names have been simplified from the actual output to aid readability.

time, for example 3886 for one of the instructions that load the current node's bounding box from the `this` pointer. The loop was the slowest part and in general most samples are on `ld` instructions, suggesting that there are many cache misses here causing memory latency to be the bottleneck. These two spots are marked as `A` and `B` respectively in figure 5.12.

```
template <typename F> void find_in_box(AABB box, F callback) const {
    if (!box.intersects(boundary)) {                    // <-- B
        return;
    }

    if (std::holds_alternative<LeafNode<Data>>(node)) {
        auto& leaf = std::get<LeafNode<Data>>(node);
        for (auto& p : leaf.points) {
            if (box.contains(p.first)) {                // <-- A
                callback(AABB(p.first, 0, 0), p.second);
            }
        }
    } else {
        auto& in = std::get<InternalNode<Data>>(node);
        in.nw->find_in_box(box, callback);
        in.ne->find_in_box(box, callback);
        in.sw->find_in_box(box, callback);
        in.se->find_in_box(box, callback);
    }
}
```

Figure 5.12: Source code of `find_in_box` for quadtree, with the problematic lines highlighted. Note that `boundary` is a field of the `Node` struct, and is thus implicitly loaded through the `this` pointer.

Checking this function with `cachegrind` confirms that theory—the above-mentioned spots had 16% and 13% L1 cache miss rates respectively, with almost no cache misses anywhere else. The LL cache miss rates in the same two spots were 61% and 16% respectively. The difference in LL miss rates makes sense, since `A` can potentially access any point stored in the tree which is a total of 16 MB in this specific test setup—larger than the 10 MB L3 cache of the processor. Meanwhile, `B` only accesses 32-byte bounding boxes for each node, which there are much fewer of since each can hold up to M points (50 in this case) if it is a leaf node, or four other nodes if it is an internal node.

The quadtree also has the `AABB::contains` method, which shows some cache misses. However, this is also called when constructing the trees and those results are hard to separate. Additionally, it would be very surprising if this callsite has any cache misses, since the point data is loaded before the call (`A`), and the bounding box data is always loaded at the start of the function (`B`).

A similar pattern can be seen for `rtree::Node::find_in_box` for both the R-tree and the Hilbert tree—the majority of the samples are concentrated in `ld` instructions within a loop that goes through all entries in a leaf node (`C` in figure 5.13). Similar to `A` in the quadtree

```
template <typename F> void find_in_box(AABB box, F callback) const {
    if (std::holds_alternative<LeafNode<Data>>(node)) {
        auto& leaf = std::get<LeafNode<Data>>(node);
        for (auto& e : leaf.children) {
            if (box.intersects(e.box)) {              // <-- C
                callback(e.box, e.data);
            }
        }
    } else {
        auto& in = std::get<InternalNode<Data>>(node);
        for (auto& e : in.children) {
            if (box.intersects(e.box)) {              // <-- D
                e.node->find_in_box(box, callback);
            }
        }
    }
}
```

Figure 5.13: Source code of `find_in_box` for R-tree and Hilbert tree, with the problematic lines highlighted.

case, these misses occur when loading the bounding box for each entry in a leaf node, with 26% L1 and 91% LL cache miss rate in case of the R-tree. There are also some misses when loading bounding boxes for internal nodes (`D`), with 16% L1 and 2% LL cache miss rates.

Interestingly, even though the Hilbert tree uses the same `find_in_box` function, the `cachegrind` results were somewhat different—`C` had 20% L1 and 39% LL misses, while `D` had 11% L1 and 6% LL misses. The most significant difference here is the lower LL miss rate, less than half of the R-tree, for `C`. It is a bit unclear why this happens, but it could be because the Hilbert tree accesses less of the tree for each query, letting frequently accessed leaf nodes (likely ones covering a large area where the points are sparse) stay in cache for longer.

The `cachegrind` tests show that cache misses are a significant factor in the performance of all three trees, and finding ways to prevent or mitigate these could be worthwhile. In particular, software prefetching could be used in strategic places, e.g. in the loops that were shown to have many cache misses, to potentially reduce the impact of these cache misses and improve performance.

Chapter 6
Discussion

6.1 M values

The M value, the radius and the number of points in the tree affect query performance and it was not feasible to test all possible combinations of them. When deciding what M values to use for each of our trees we therefore had to make some simplifications. In order to eliminate the M value as a parameter for the rest of our tests we wanted to find a "compromise" value that is good for different kinds of data.

As mentioned we used two different seeds in the testing as a sanity check. One could argue how we know the chosen seeds aren't two special cases, however, we find it very unlikely that it would be the case. We also checked the plots and they behaved like expected, based on our knowledge of the trees. We also checked that the two plots looked similar, if they would differ noticeably we would test it more thoroughly.

In practice, the difference between different M values ended up being smaller than expected, meaning that our results are probably not far off from what you would get if you optimize the M value for your specific application.

A larger dataset means a more dense set. This is because the surface area of the dataset is limited to the Earth's surface, i.e. latitude is between $[\frac{-\pi}{2}, \frac{\pi}{2}]$ and longitude is between $[-\pi, \pi]$. Since all of these tests used the same query radius (100 km), the number of points returned was much larger for the larger datasets. This means that for the large dataset, a lot of the fine structure in the trees just end up adding overhead to our queries, since e.g. for most leaf nodes every single point will end up being returned anyway. This is likely the reason why the large dataset is not slower with larger M values—in the deepest levels of the tree larger M values are beneficial since we are able to loop through lots of points that are stored contiguously in memory and are in many cases all returned. It is likely that some performance is lost in the top levels of the tree for such high M values, but since less time is spent there the previous factor ends up being more significant.

As already established the M value is the number of children an internal node can hold.

This means that the M value is one of the key factors to the speed of the trees. Since the M value affects the depth and width of the tree the M we are searching for in these tests is the one that finds a balance in the tree.

When looking at the graphs in 5.1 all plots show that the M cannot be too small. This is quite intuitive because a small M gives a deep tree, and since the quadtree is unbalanced the tree can get very deep. A deep tree means a longer search time because every pointer-step that has to be gone through, hence a bad search tree. The search time also increases because when the leaf nodes contain few data points, the search algorithm has to search trough more leaf nodes.

We also saw that a larger M performed worse and this could be the result of too many data points in every leaf node meaning that when the right leaf node is found the search through the array holding all data points takes too long. For the larger dataset this seems to not be a problem but that may be because the set is too large meaning that it's still faster to perform a linear search than go deeper in the tree.

The R-tree follows the same idea, however the results are somewhat different. Here the M affects all internal nodes meaning that the tree can get very wide but also very deep. Our tests show that a small M is bad, see figure 5.2, again since it will be very deep and a deep tree takes a long time to search through. However, there is no clear indication that a large M is bad, as one would intuitively think.

The Hilbert tree had some more shocking results. Since the data sorting is much better, the M is more the balance of a deep versus wide tree. The results from figure 5.3 show quite clearly in the smaller datasets that a deep tree is better and that a linear search takes more time. However in the large dataset it doesn't really matter what M is chosen, which could indicate that as the tree reaches a certain size it doesn't matter whether it is wider or deeper.

Over all, for values larger than 20, the M doesn't make that much of a difference in execution time for the R-tree and quadree. Even though all plots look quite dramatic, the time step is not as significant as it looks. For the Hilbert tree this applies even more because the time step difference is very small.

6.2 Varying the number of points

Three different tests were performed to find the best search tree. The first test evaluated each tree by containing an increasing number of both uniformly distributed and geographic data points using the `is_inside_radius()` function, and one without. We also tested to scale the radius to the dataset, i.e statistically for every test the tree would find the same number of data points.

To get a better understanding of the search trees' performance and not the filter function's performance, we also tested the search trees on an increasing number of geographic data points without the `is_inside_radius()` function. In this test using geographic data it was still obvious that the R-tree is bad, however the difference between the quadtree and Hilbert tree is still not huge, especially not with a small number of data points. The difference in performance is after 10^5 data points where the Hilbert tree is faster than the other two. However, within the interval 10^3 and 10^5 the quadtree is marginally better. This means that for the purpose of the search tree the quadtree may be better, since the dataset that is used contains around 10^4 data points.

When testing an increasing number of uniform data points the graph is similar to the geographic data without `is_inside_radius()` test. There is a larger difference between the trees than geographical and the results are quite interesting. When using irrelevantly small datasets containing 1 point and 100 points, the quadtree seems to have the worst performance which is very surprising considering earlier results. Over all in figure 5.7 the Hilbert tree has the best and most reliable performance. However, as mentioned earlier the search tree is supposed to be used on the dataset given by the company, containing around 10^5 data points, meaning that the quadtree might still be the better alternative.

The third test that is seen in figure 5.4 shows some unexpected results. This test gives a better comparison of the three structures because of the scaled search radius. The plot shows that for smaller datasets (10 —100 points) there is not much difference between the search trees. However, after 10^3 we can see how the R-tree starts to perform worse than the other trees, and after 10^4 the runtime increases exponentially. There could be many reasons for this, but we think the most logical reason for this is the structure of the R-tree. When the dataset is small, finding information is quick regardless of the way it is stored. However, when the data size is increased the search tree's structure and performance is of importance. In the plot it becomes obvious that the R-tree's design is suboptimal. The initialization of the R-tree constructs overlapping bounding boxes due to the way the data is grouped, and this makes the search very slow.

6.3 Varying the radius

To test how the search radius affected the performance we fixed the number of points and increased the search radius. The results were similar to earlier tests, the R-tree performed poorly and the other two trees were almost equal, but now with the quadtree as the best until when the radius was 10^5 m where the Hilbert tree performs marginally better.

Again, for the purpose of the search tree the only relevant radii are between 10 km and 1000 km (10^4–10^6 m), which means that over all the quadtree and the Hilbert tree are interchangeable. However, considering it will be used for a flight map, a larger radius will probably be used more meaning that the Hilbert tree most likely is the best for this parameter.

Once again the R-tree is by far worst, and this could be because of the way it is structured. As explained before, the tree's structure is depending on the order of the inserted points and since the data isn't sorted in any particular way the structure of the R-tree is, as seen in the results, very bad. The Hilbert tree and quadtree are better structured meaning that they are faster even though the radius is increasing.

6.4 Time complexity

When increasing the number of points (N) in the dataset and keeping everything else constant (including the query area), the number of points returned by an individual query goes up proportionally. This can be clearly seen in figures 5.4a, 5.5a, 5.6a, and 5.7a, where the plots all look approximately like straight lines for large numbers of points. This is true for all three trees, albeit with different constant factors for each tree in the benchmarks where `is_inside_radius` was not used, including the ones using uniform data. The empirical

results seem to indicate that the average-case time complexity is $O(N)$, although is would be nice to confirm this with theoretical analysis.

In the test with scaling radius, the query area is decreased in proportion to the increase in N, in order to keep the expected number of points returned the constant. Despite this, we still see the plots slope upwards in figure 5.8, indicating that these two factors do not completely cancel out in the execution time for queries. Quadtrees and Hilbert trees show a very slight upwards slope—presumably there is a $O(\log N)$ relationship between the dataset size and the execution time in this case, which make sense given that the depth of the trees is proportional to $\log N$. In the tests without scaling radius, this effect is likely overshadowed by other factors, e.g. the large number of points that must be processed and returned, and thus not visible. R-trees look very bad here, definitely worse than $O(\log N)$ and possibly even as bad as $O(N)$.

6.5 Point-in-polygon evaluation

The trees used for the polygon problem were tested on two datasets, one from the hosting company and one random subset from OpenStreetMap. As seen in the tables, the Hilbert tree was faster than the R-tree. However, the difference in performance is a lot less than we had anticipated. When looking at the `operf` results, it can be seen that the majority of the time taken is the `is_inside_polygon()` function.

There could be many reasons for the even results. It is most likely due to the fact that `is_inside_polygon()` is disproportionally slow. For all tests the `is_inside_polygon()` takes the majority of the time meaning that the performance of the actual trees is not really shown. If the `is_inside_polygon()` function is taken away the difference might be a lot bigger between the trees' performances.

6.6 Implementation complexity

Apart from testing execution time and memory usage, there are other factors in choosing the best data structure. A more complex data structure provides more opportunities for bugs and/or vulnerabilities to arise, although in a stable and well-tested library this might not be a large concern.

Again, the R-tree is the worst alternative because of the complicated structure. The insertion process is dynamic meaning that when points are inserted the tree is dynamically restructured to not exceed the M value. This means that there are a lot of helper functions in the construction of the R-tree, including many recursive functions, pointer manipulation and invariants that need to be maintained. This results in a tree that is complex to implement and hard to understand.

The Hilbert tree may be the tree that is the most conceptually difficult to understand. Since it is a type of R-tree, it has the same structure. The fact that it also uses the Hilbert curve is what makes it complicated. The Hilbert curve is hard to understand, but mostly complex to translate to code. However, when the Hilbert curve is implemented the rest of the construction is quite straight forward. The static, bottom-up construction means that an `insert` function is unnecessary, as we never need to insert points dynamically. This reduces

the amount of code needed significantly and is generally easier to reason about in our opinion.

In this regard, the quadtree was the best option. The quadtree has a simple construction and has some similarities with the binary search tree, meaning that it over all is very simple to understand and implement. Unlike the R-tree, there aren't a lot of helper functions or dynamic restructuring, so it is much easier to understand than the Hilbert tree.

As mentioned in section 3.1, our C++ code made heavy use of templates. This somewhat complicated our analysis with `oprofile`, as `opannotate -s` was not able to reliably map instructions from the binary to lines in the source file, resulting in most of the relevant data being assigned to "line 0". It is possible that this could be resolved by finding the right combination of compiler version, `oprofile` version and/or compiler flags, but we were not able to do so. Instead we used `opannotate -a` to look directly at the assembly, and manually matched the code in some interesting spots to their corresponding lines of code. Then we used `cachegrind` to verify our findings.

6.7 Comparison to other works

Article [11] compares the performance of the quadtree and the R-tree as spatial indexes in spatial databases. In the study the authors implemented the linear quadtree and a combination of different initializations of the R-tree, and compared these. The areas compared were insertion of different types and sizes of information datasets (point, line segments and polygons), updating information and searching for different types of queries. The results showed that the efficiency of inserting points depended on the type of information being inserted and the size of the datasets. The R-tree was faster for large polygon datasets and the quadtree was faster for smaller sized datasets and for point data. When testing different queries the R-tree outperformed the quadtree in every test. As for memory usage the article's quadtree again needed more than the R-tree.

The results gotten in the article are quite different compared to our results. In almost all our cases the quadtree was superior to the R-tree, which was not the case for the article. Even for the memory usage we got different results.

It's questionable whether it is valid to compare these results to the ones gotten in this thesis. The article [11] uses different types of structures, especially for the R-tree where they use a combination of different initializations compared to the Guttman R-tree used in this project. We also focused the testing on benchmarking the search time and not the construction time.

In article [9] the authors present the Hilbert packed R-tree, the implementation and compare it to three other types of R-trees: the quadratic split Guttman R-tree, the R-tree and the `lowx` R-tree. The trees were all tested for query time for different datasets, where the results gotten were that the Hilbert tree was superior to the other types of R-trees.

These results align well with our results for the Hilbert tree compared to the R-tree. Since we used this article as reference to implement the Hilbert tree, the Hilbert trees should be mostly the same. The R-tree used are the same in both implementations as well, i.e the quadratic split Guttman R-tree. The results in article [9] give a good cross reference to this thesis and could be used to validate the results we got. It also confirms that we implemented the Hilbert tree and the R-tree in a similar way to the original implementations, suggesting that our results are reliable.

6.8 Sources of error

There are a few areas that may cause our results to be unreliable. One source of error could be the limited amounts of datasets used. In this thesis we constructed most datasets ourselves, and the datasets given were limited. This means that some special cases weren't tested and the search trees were not tested to their limited capacity. If they were more thoroughly tested, the outcome may have been different. One thing that we could have tried was to use the same datasets as the ones used in the original implementation of the trees. However, there was no way to find these, they were not stated in the reports or described anywhere. Therefore these datasets could not be used.

Another source of error is the approximation made that the Earth is a perfect sphere. As mentioned in the theory section, the Earth is not perfectly round but somewhat flattened around the poles. In our calculations we approximated the Earth to be a sphere to make the calculations easier, however this could lead to a slight difference between our results compared to the real world—our circles end up being slightly distorted compared to ideal circles. It could be discussed whether this error actually would make an impact on which that is the fastest, either way the results would be more accurate.

As mentioned in section 5.6, quadtrees have a slight unfair advantage in our evaluation due to the decision to only store points (16 bytes) in their leaf nodes, as opposed to bounding boxes (32 bytes) which are stored in R-trees. This has a big impact on memory usage, of course, but it could also affect execution time as more data needs to share a limited number of cache lines, possibly leading to more cache misses. Ideally these benchmarks would have been run with a specialized version of R-trees that store points instead of bounding boxes, but as we realized this too late in the process, we opted to note it here instead.

Although we store all coordinates as doubles in our code, all trigonometric operations are performed using 32-bit floating point values for performance reasons. Since 32-bit floats have approximately 7 decimal digits of precision and the Earth's circumference is around 2×10^7 m, the results have an error of up to around 2 meters. Since we are dealing with radii on the order of 10^5 m (100 km), this is not a big deal and should not meaningfully affect the results. In fact, this precision is probably overkill for the host company's application.

We did not use any statistical analysis in our measurements, making it hard to reason about the uncertainty in a rigorous way. Ideally, each benchmark would have been run multiple times with different seeds each time for both the random datasets and the test cases, and error bars or similar added to our plots.

The fact that we only tested on one computer limits the generalizability of our results somewhat—we would not expect the overall shape of the plots to be different, and e.g. R-trees are far enough behind that they are almost certainly worst regardless of where the code is run, but quadtrees and Hilbert trees are often close enough in performance that which one ends up being faster could possibly depend on the execution environment.

6.9 Future work

This thesis gives a comparison of the three search trees—the quadtree, the R-tree, and the Hilbert packed R-tree. However, there are many opportunities for further research.

The most obvious topic would be to compare a wider range of search trees. We limited

our scope to just a few different trees that were reasonable to implement in the time frame we had. If we had more time, some other trees that would be interesting to try would be the R*-tree, the octree and the dynamic Hilbert tree.

The R*-tree is a very popular R-tree variant, promising superior performance. It would be interesting to check if it is the best performing search tree or if a better alternative exists.

The octree is intriguing since it is a three-dimensional analog of the quadtree. Having all points in 3D space would eliminate distortion and remove the need for special handling around the poles. It would remove the need for some coordinate conversions that take significant time in our current implementation, potentially improving performance. One possible disadvantage is that the points are quite sparse in 3D space since they are only on the surface of the Earth, which makes large parts of the octree empty and could make it deeper than a quadtree would be for the same dataset. It is not obvious that it would be faster in practice, but it would be interesting to study further.

Both the R-tree and the Hilbert tree can easily be generalized to three dimensions, which would also be worth researching.

The dynamic Hilbert tree would be interesting to examine because the article [10] claim that this structure is superior to all other existing search trees. It would be interesting to investigate if that claim is true.

This thesis compares quite basic implementations of each search tree, to build on this a future work could enhance the search trees' structures. An example of this could be to construct the Hilbert trees to be more even. When building the Hilbert tree, if M is 5 and there are 6 nodes to be grouped, the current implementation will construct two nodes, one containing 5 nodes and one containing 1 node. To improve this suboptimal build, one could make the division of nodes more equal by e.g. constructing two nodes containing 3 nodes each.

Another obvious modification that could change the results is to not approximate the Earth to a perfect sphere. This simplification makes the circles slightly warped if placed on the real Earth, but since the Earth is *almost* a sphere the difference should not be very large. Correcting this would likely make `is_inside_radius` slightly slower, but is unlikely to affect our results significantly.

Another interesting structure to implement is a grid without a tree structure. Instead of dividing the dataset as done in the quadtree, it would be interesting to construct a grid structure where every cell has a fixed size. To search in this structure one would simply have to look at all tiles that intersect the query region. If the tiles are appropriately sized each query might only have to look at a few of them, reducing overhead.

As mentioned in section 6.8, the precision of the standard library's trigonometric functions is probably overkill for the application considered in this report. Future research could involve trying different approximations for these functions and comparing both their performance and how much they affect the results. When using large radii you could probably reduce the precision a lot before the results change significantly.

While profiling our program we noted that cache misses were responsible for a large proportion of the execution time. One way to mitigate this could be to use prefetch intrinsics to load data into memory before it is needed. For this to be effective we (the programmers) have to be able to predict what memory is to be loaded ahead of time, which should be possible for our `find_in_box` functions. For example, in each of the loops containing problematic load instructions (see figures 5.12 and 5.13), prefetch the data that is needed one or more iterations

ahead of the current one. We did not have time to try this, but it could potentially result in a noticeable speedup.

There are opportunities to make space-time tradeoffs in our code by precomputing the results of expensive trigonometric functions, e.g. converting each point to Cartesian coordinates ahead of time and storing them. This would save a lot of computation but also likely increase the cache miss rate because of more memory being accessed regularly. One could investigate whether this is worth the tradeoff or not.

Chapter 7
Conclusion

Overall, our results were quite different from expectation. Our initial thoughts based on the references were that the quadtree would be the slowest tree, the R-tree would be slightly better and the Hilbert tree would be the best. Instead, the quadtree and the Hilbert tree performed almost equally well for the find-in-radius tests while the R-tree was unexpectedly slow. For almost all middle sized datasets, and some of the smaller datasets, the quadtree performed the best. The Hilbert tree outperformed the others on larger datasets and also on some of the smaller datasets.

Also, more in line with expectation, all trees were not able to handle polygons. Since only the R-tree and Hilbert tree could handle polygons the way they were originally implemented, only these were tested. Again the Hilbert tree performed better than the R-tree. This time the difference in runtime was more subtle compared to the radius tests. However, as mentioned in the discussion, this could be a result of a slow `is_inside_polygon()` function.

To summarize the findings the Hilbert tree was generally the best structure, it is compatible for both areas tested in this report and the tree was not too complex to implement. For the company we worked with, we would however recommend to use the quadtree for the find-in-radius problem, especially since the quadtree was the best option when the company's dataset was tested.

7.1 Research questions

In this thesis we have tried to find which search tree is the best: quadtree, R-tree or Hilbert tree, given following tasks:

- Provided a set of points (locations on the Earth's surface), find all points within a given radius of a given point
- Provided a set of polygons (representing time zones), find which polygon that a given point is inside of.

To evaluate, we have answered two research questions.

- *What are the advantages and disadvantages for each data structure, considering execution time and memory usage, given the tasks above?*

  Considering the quadtree, the main disadvantage is inability to handle polygons. The point-region quadtree could be modified to be compatible with polygons, however it would not remain the same PR structure, which means it would no longer be comparable as a PR quadtree. Apart from this, the tree is surprisingly fast at finding the points within the radius of a given point, regardless of if the number of data points change or the radius size increase.

  The R-tree was a disappointment, being the slowest of all search trees in every aspect. For very small datasets it performed similarly to the other trees, but it quickly escalated in execution time. The only advantage was the fact that it could solve both problems, the radius and the polygon problem. The performance for polygon search was still slower than the Hilbert tree though. Over all the R-tree is a suboptimal search tree.

  The Hilbert tree was superior to the other search trees. It could handle both problems and it was almost always the fastest. There were no real disadvantages with this search tree, it is a stable and fast alternative. It performed well for larger datasets, and was marginally slower than the quadtree for medium sized datasets.

  As the trees were implemented for our benchmarks, the quadtree had a big advantage in terms of memory usage. However, if you put the R-tree and Hilbert tree on equal footing with the quadtree by storing points instead of bounding boxes in the leaf nodes, the conclusion is different. The memory usage ends up depending mostly on the M value regardless of the type of tree, with the Hilbert tree being the worst, the quadtree in the middle, and the R-tree the best in this regard. The difference between these is on the order of 30%, which might matter if you are very memory constrained but we would expect that the execution time is a bigger factor for most users.

- *Is one of the data structures superior or does it depend on situation? How does the data size affect this?*

  For the find-in-radius case, there is no obvious best structure. When increasing the data size, the quadtree and Hilbert tree are both performing well. The quadtree is a little better for smaller datasets while the Hilbert tree is faster for larger datasets.

  When considering the implementation complexity, the quadtree is superior. The quadtree has a simple structure easy to implement. The Hilbert tree on the other hand, is more complex. However, compared to the R-tree the Hilbert tree is simpler since the R-tree uses a lot of help functions.

  Overall, the Hilbert tree is by far superior in the general case. It performed well in all tests, it is compatible with polygons and is fairly easy to implement. It is stable, meaning that it performs similarly relative to the dataset size. It was the best performing search tree for larger datasets in the find-in-radius problem and for all tests in the polygon problem. The quadtree was the best for smaller datasets and when considering the implementation complexity.

References

[1] Carl Johan Balck. User interaction in inflight entertainment map application. Master's thesis, Lund University, 2019.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard 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*, pages 322–331, 1990.

[3] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[4] Encyclopaedia Britannica Editors. Latitude and longitude, 2025. Accessed 10 September 2025.

[5] Raphael A. Finkel and Jon L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[6] W. Randolph Franklin. Pnpoly: Point inclusion in polygon test, 1999. Accessed 14 November 2025.

[7] Ralf Hartmut Güting. An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399, 1994.

[8] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Record*, 14(2):47–57, 1984.

[9] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *Proceedings of the Second International Conference on Information and Knowledge Management (CIKM)*, pages 490–499. ACM, 1993.

[10] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 500–509, 1994.

[11] Ravi Kanth V. Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and r-tree indexes in oracle spatial: A comparison using gis data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 546–557. ACM, 2002.

[12] Loyola Marymount University. Search trees. Accessed 16 September 2025.

[13] Takashi Matsuyama, Makoto Nagao, et al. A file organization for geographic information systems based on spatial proximity. *Computer Vision, Graphics, and Image Processing*, 26(3):303–318, 1984.

[14] OpenDSA Project. The pr quadtree. Accessed 16 September 2025.

[15] OpenStreetMap Foundation. Welcome to openstreetmap, 2025. Licensed under CC BY-SA 2.0.

[16] Osmcode Project. Libosmium: A fast and flexible c++ library for working with openstreetmap data, 2025. Licensed under the Boost Software License.

[17] Queen's University, School of Computing. Ray tracing acceleration methods: Axis-aligned bounding boxes (aabb), 2025. Lecture notes for CISC 454.

[18] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[19] John Skilling. Programming the hilbert curve. *AIP Conference Proceedings*, 707(1):381–387, 2004.

Appendices

Appendix A
Tables

This chapter presents tables showing results from testing. The tables appear in the same order as the results are presented in the results chapter.

Quadtree—M values

M	Small 1	Small 2	Medium 1	Medium 2	Large 1	Large 2
10	0.7797	0.7673	22.3202	22.2910	2112.73	2083.74
20	0.7327	0.7305	21.5491	21.5255	2025.77	2025.70
30	0.7286	0.7262	21.0223	21.0036	1962.86	1963.14
40	0.7252	0.7228	20.9970	20.9424	1918.84	1919.33
50	0.7291	0.7269	20.9048	20.8905	1903.62	1903.55
60	0.7355	0.7334	20.8768	20.8694	1899.64	1900.15
70	0.7406	0.7398	20.8790	20.8838	1896.74	1896.71
80	0.7430	0.7420	20.9312	20.9303	1892.48	1893.07
90	0.7430	0.7429	21.0461	21.0388	1889.40	1889.81
100	0.7436	0.7429	21.1605	21.1698	1886.65	1886.75
110	0.7462	0.7429	21.2369	21.2419	1883.89	1884.01
120	0.7463	0.7470	21.2505	21.2548	1880.69	1881.10
130	0.7528	0.7510	21.2586	21.2664	1877.59	1877.87
140	0.7651	0.7675	21.2688	21.2801	1873.68	1872.74
150	0.7762	0.7758	21.2905	21.2982	1870.11	1869.54

Table A.1: Execution time for quadtree queries with different M values on three sizes of dataset, measured in µs. The queries return all points within 100 km of a given point. Each benchmark was run with two different seeds.

R-tree—M values

M	Small 1	Small 2	Medium 1	Medium 2	Large 1	Large 2
10	1.1461	1.1652	43.8291	43.8238	3393.19	3128.10
20	1.2987	1.2822	41.9370	39.8633	2760.14	2731.17
30	1.1673	1.1077	34.0183	32.1276	2477.15	2450.13
40	1.0991	1.2717	34.9743	33.7014	2446.90	2390.22
50	1.1494	1.1692	31.5112	31.5558	2298.90	2317.23
60	1.1588	1.1101	29.5608	28.8331	2238.26	2216.00
70	1.2127	1.2733	30.1542	32.2188	2261.25	2316.21
80	1.2631	1.2584	30.8339	31.2637	2214.40	2297.00
90	1.1767	1.2470	28.6596	28.6995	2196.94	2205.42
100	1.2250	1.3051	29.6835	29.6495	2205.30	2199.80
110	1.3779	1.4814	31.5742	30.1982	2161.51	2211.39
120	1.3015	1.3676	29.9818	30.4565	2204.47	2228.47
130	1.4428	1.4533	30.2277	30.5081	2192.91	2174.60
140	1.3682	1.4420	29.3889	30.2522	2178.06	2160.97
150	1.3953	1.4586	29.3075	29.7630	2093.40	2147.51

Table A.2: Execution time for R-tree queries with different M values on three sizes of dataset, measured in µs. The queries return all points within 100 km of a given point. Each benchmark was run with two different seeds.

Hilbert tree—M values

M	Small 1	Small 2	Medium 1	Medium 2	Large 1	Large 2
10	0.7820	0.7854	20.5934	20.6115	1848.26	1847.85
20	0.8287	0.8369	20.5509	20.5662	1845.76	1845.85
30	0.8637	0.8604	20.5480	20.5781	1846.80	1846.88
40	0.9281	0.9318	20.6708	20.6962	1847.12	1847.30
50	0.8969	0.9051	20.7493	20.7874	1849.03	1847.99
60	0.9950	0.9984	21.0017	21.0498	1855.62	1858.87
70	1.0638	1.0605	20.9555	20.9894	1845.76	1846.07
80	1.1146	1.1336	21.0382	21.0696	1846.28	1846.52
90	1.1379	1.1510	21.1422	21.1798	1847.49	1847.69
100	1.1591	1.1684	21.2006	21.2468	1846.02	1846.13
110	1.1764	1.1859	21.2352	21.2617	1846.07	1845.99
120	1.1978	1.1998	21.2948	21.3303	1846.35	1846.38
130	1.2073	1.2206	21.3592	21.3839	1846.68	1846.65
140	1.2175	1.2446	21.4305	21.4577	1846.48	1846.34
150	1.2574	1.2580	21.5188	21.5451	1846.98	1847.01

Table A.3: Execution time for Hilbert tree queries with different M values on three sizes of dataset, measured in µs. The queries return all points within 100 km of a given point. Each benchmark was run with two different seeds.

Number of points—geographic data

Nbr of points	Quadtree	R-tree	Hilbert tree
10^0	0.1021	0.1008	0.1004
10^1	0.1143	0.1186	0.1182
10^2	0.1850	0.2254	0.1890
10^3	0.3242	0.4685	0.3458
10^4	0.7174	1.2690	0.7795
10^5	2.8978	5.2100	2.9475
10^6	21.0676	31.5307	20.7721
10^7	196.3811	257.8631	189.5936
10^8	1903.1578	2232.1425	1851.4954
10^9	19329.023	21241.164	18422.293

Table A.4: Execution time in µs of queries on each tree for different numbers of random geographic data points.

Number of points—OSM

Nbr of points	Quadtree	R-tree	Hilbert tree
10^0	0.1042	0.1030	0.1032
10^1	0.1164	0.1201	0.1202
10^2	0.1845	0.1868	0.1664
10^3	0.2982	0.3671	0.2700
10^4	0.5666	0.8883	0.5938
10^5	2.2702	3.5869	2.3238
10^6	18.2009	23.9247	17.7869
10^7	174.2712	200.0996	167.9753
10^8	1728.3371	1850.6532	1659.5592
10^9	17221.709	17738.857	16547.160

Table A.5: Execution time in µs of queries on each tree for different numbers of OpenStreetMap data points.

Number of points—geographic data without `is_inside_radius()`

Nbr of points	Quadtree	R-tree	Hilbert tree
10^0	0.1023	0.1013	0.1014
10^1	0.1148	0.1185	0.1185
10^2	0.1844	0.2238	0.1876
10^3	0.3058	0.4459	0.3260
10^4	0.5170	1.0626	0.5720
10^5	1.0261	3.3423	1.0760
10^6	2.9740	13.5350	2.7205
10^7	17.0771	77.4247	10.3056
10^8	113.0279	440.7718	60.2583
10^9	1350.9029	3728.7988	489.5275

Table A.6: Execution time in µs of queries on each tree for different numbers of random geographic data points, excluding the `is_inside_radius()` function.

Different number of points—uniform data

Nbr of points	Quadtree	R-tree	Hilbert tree
10^0	0.0074	0.0067	0.0068
10^1	0.0184	0.0222	0.0224
10^2	0.0723	0.1134	0.0867
10^3	0.1414	0.2557	0.2012
10^4	0.2405	0.5860	0.3555
10^5	0.3744	1.6339	0.5607
10^6	0.8672	5.8743	1.0382
10^7	2.6921	25.7638	2.2378
10^8	10.5231	118.5765	6.3034
10^9	81.0858	782.2028	32.1988

Table A.7: Execution time in µs of queries on each tree for different numbers of random uniform data points.

Different radii—geographic data

Radius	Quadtree	R-tree	Hilbert tree
10^0 m	0.0729	0.484	0.0855
10^1 m	0.0733	0.484	0.0854
10^2 m	0.0734	0.485	0.0855
10^3 m	0.0748	0.495	0.0872
10^4 m	0.110	0.615	0.123
10^5 m	2.12	3.19	2.08
10^6 m	230	238	226
10^7 m	19300	19400	18900
2.002×10^7 m	21800	21800	21200

Table A.8: Execution time in µs of queries with different radii for each tree on a set of 10^6 random geographic data points. The last row represents half the Earth's circumference, i.e. the largest possible meaningful radius, which always returns all points.

Memory usage

	Random geographic			Random uniform			OpenStreetMap		
Points	Quadtree	R-tree	Hilbert	Quadtree	R-tree	Hilbert	Quadtree	R-tree	Hilbert
10^0	104 B	104 B	104 B	104 B	104 B	104 B	104 B	104 B	104 B
10^1	320 B	464 B	464 B	320 B	464 B	464 B	320 B	464 B	464 B
10^2	2.77 kB	4.22 kB	4.86 kB	2.77 kB	4.22 kB	4.86 kB	3.06 kB	4.22 kB	4.86 kB
10^3	27.8 kB	41.4 kB	48.9 kB	30.1 kB	41.4 kB	48.9 kB	30.1 kB	41.6 kB	48.9 kB
10^4	285 kB	415 kB	489 kB	267 kB	415 kB	489 kB	291 kB	415 kB	489 kB
10^5	2.74 MB	4.15 MB	4.89 MB	2.79 MB	4.15 MB	4.89 MB	2.92 MB	4.15 MB	4.89 MB
10^6	28.5 MB	41.5 MB	48.9 MB	29.9 MB	41.5 MB	48.9 MB	28.9 MB	41.5 MB	48.9 MB
10^7	287 MB	415 MB	489 MB	267 MB	415 MB	489 MB	288 MB	415 MB	489 MB
10^8	2.74 GB	4.15 GB	4.89 GB	2.81 GB	4.15 GB	4.89 GB	2.87 GB	4.15 GB	4.89 GB
10^9	28.5 GB	41.5 GB	48.9 GB	29.9 GB	41.5 GB	48.9 GB	28.8 GB	41.5 GB	48.9 GB

Table A.9: Memory usage for different trees, datasets, and numbers of points.

Appendix B
Pseudocode

B.1 Common code

```
/// Converts a point to 3D Cartesian coordinates,
/// with the center of the Earth as the origin.
/// Assumes that p.y is the latitude and p.x the longitude.
CartesianPoint to_cartesian(p):
    phi = p.y * PI / 180
    lambda = p.x * PI / 180
    return CartesianPoint(
        x: cos(phi) * cos(lambda),
        y: cos(phi) * sin(lambda),
        z: sin(phi),
    )

/// returns whether the two points are within ‘radius‘ meters
/// of each other on the Earth’s surface
bool is_inside_radius(a, b, radius):
    phi = radius / 6371000
    a_c = to_cartesian(a)
    b_c = to_cartesian(b)
    return phi >= acos(mid_c.x * p_c.x
                     + mid_c.y * p_c.y
                     + mid_c.z * p_c.z)
```

```
/// Finds an approximate bounding box for a circle on the Earth's surface.
AABB find_bbox(mid, radius):
    phi = mid.y * PI / 180 // latitude in radians
    r = radius / 6371000   // circle radius in radians
    if abs(phi) + r >= PI / 2:
        // the circle goes through one of the poles
        w = 2 * PI
        h = PI / 2 - abs(phi) + r
        res_phi = sign(phi) * (PI / 2 - h / 2)
    else:
        max_width_phi = abs(phi) + r
        w = 2 * r / cos(max_width_phi)
        h = 2 * r
        res_phi = phi
    return AABB(
        center: Point(
            x: mid.x,
            y: res_phi * 180 / PI,
        ),
        width: w,
        height: h,
    )

/// Finds all points within a certain radius of the center
/// and calls the callback for each.
find_in_radius(tree, center, radius, callback):
    box = find_bbox(center, radius)
    find_in_box_wrapping(tree, box, (bb) -> {
        if is_inside_radius(center, bb.center, radius):
            callback(bb)
    })
```

```
/// This function handles wrapping the longitude correctly
/// when a bounding box extends outside the range [-180, 180].
/// assumes that the tree has a find_in_box function
/// the callback is called for each point inside the given box
find_in_box_wrapping(tree, box, callback):
    if box.width >= 360:
        box.width = 360
        box.center.x = 0

    // x of left side of box, with the boundary starting at x=0
    left_edge = mod(box.center.x - box.width / 2 + 180, 360)

    // amount of the box that wraps around from the right to the left edge
    rest = left_edge + box.width - 360

    if rest > 0:
        // non-wrapping part
        non_wrapping_width = 360 - left_edge
        find_in_box(tree, AABB(
            center: Point(
                x: left_edge + non_wrapping_width,
                y: box.center.y,
            ),
            width: non_wrapping_width,
            height: box.height,
        ), callback)

        // wrapping part
        find_in_box(tree, AABB(
            center: Point(
                x: rest / 2 - 180,
                y: box.center.y,
            ),
            width: rest,
            height: box.height,
        ), callback)
    else:
        // box does not wrap
        find_in_box(tree, AABB(
            center: Point(
                x: left_edge + box.width / 2,
                y: box.center.y,
            ),
            width: box.width,
            height: box.height,
        ), callback)
```

B.2 Quadtree

```
/// Finds all points within the box
/// and calls the callback for each one.
find_in_box(node, box, callback):
    if !box.intersects(node.boundary):
        return

    if node is a LeafNode:
        for p in node.points:
            if box.contains(p):
                callback(p)
    else:
        find_in_box(node.nw, box, callback)
        find_in_box(node.ne, box, callback)
        find_in_box(node.sw, box, callback)
        find_in_box(node.ne, box, callback)

/// Insert a point into the quadtree,
/// if it is within the bounds of the given node.
insert(node, p):
    if !boundary.contains(p):
        return

    if node is a LeafNode:
        if node.points.size() == M:
            // subdivide
            points = node.points

            node = create a new InternalNode with four LeafNodes
                   as children

            // reinsert old points
            for q in points:
                insert(node, q)

            // insert new point
            insert(node, p)
        else:
            node.points.push(p)
    else:
        match direction_from(node.boundary.center, p):
            NW -> insert(node.nw, p),
            NE -> insert(node.ne, p),
            SW -> insert(node.sw, p),
            SE -> insert(node.se, p),
```

B.3 R-tree

```
/// Finds all points within the box
/// and calls the callback for each one.
find_in_box(node, box, callback):
    if node is a LeafNode:
        for e in node.children:
            if box.intersects(e.box):
                callback(e.box, e.data)
    else:
        for e in node.children:
            if box.intersects(e.box):
                find_in_box(e.node, box, callback)

Node choose_leaf(tree, bb):
    N = tree.root
    loop:
        if N is a LeafNode:
            return N
        else:
            least_increase = -∞
            best_child = -1
            for i in [0, N.children.size()):
                child = N.children[i]
                area_before = child.box.area()
                area_after = child.box.expand(bb).area()
                diff = area_after = area_before
                if diff < least_increase:
                    least_increase = diff
                    best_child = i
            N = node.children[best_child]
```

```
// LeafEntry or InternalEntry
template<Entry>
(vector<Entry>, vector<Entry>) split_node(in):
    // pick seeds (two nodes that should NOT be in the same branch)
    worst_area = -∞
    worst_i = -1
    worst_j = -1
    for i in [0, in.size()-1):
        for j in [i+1, in.size):
            J = combine boxes in[i].box and in[j].box
            d = J.area() - in[i].box.area() - in[j].box.area()
            if d > worst_area:
                worst_area = d
                worst_i = i
                worst_j = j

    vector<Entry> a
    vector<Entry> b
    a.push(in[worst_i])
    b.push(in[worst_j])

    remove the pushed nodes from in

    a_box = a[0].box
    b_box = b[0].box

    while !in.empty():
        if a.size() + in.size() == m:
            push the rest of the nodes to a
            break
        else if b.size() + in.size() == m:
            push the rest of the nodes to b
            break

        max_diff = -∞
        max_i = -1
        put_in_b = false
        for i in [0, in.size()):
            a_before = a_box.area()
            a_after = a_box.expand(in[i].box).area()
            d1 - a_after - a_before

            b_before = b_box.area()
            b_after = b_box.expand(in[i].box).area()
            d2 - a_after - a_before
```

```
            diff = abs(d2 - d1)
            if diff > max_diff:
                max_diff = diff
                max_i = i
                put_in_b = d2 < d1

        if put_in_b:
            b_box = b_box.expand(in[max_i].box)
            b.push(in[max_i])
        else:
            a_box = a_box.expand(in[max_i].box)
            a.push(in[max_i])
        remove element i from in

    return (a, b)

/// Calculates the smallest bounding box that can fit all
/// children of the given node.
AABB calc_bbox(node):
    res = node.children[0].box
    for child in node.children:
        res = res.expand(child.box)
    return res

Node adjust_tree(tree, L, LL):
    N = L
    NN = LL

    while N != tree.root:
        P = N.parent
        EN = entry in P.children corresponding to N
        // recalculate bounding box
        EN.box = calc_bbox(N)

        if NN != null:
            P.children.push(InternalEntry(calc_bbox(NN), NN))
            if P.children.size() > M:
                a, b = split_node(P.children)
                PP = MiddleNode(b, P.parent)
                for child in PP.children:
                    child.parent = PP
                NN = PP
            else:
                NN = null
        N = P
```

```
insert(Rtree tree, AABB bb):
    L = choose_leaf(tree.root, bb)
    LL = null

    L.children.push(bb)
    if L.children.size() > M:
        a, b = split_node(leaf.children)
        leaf.children = a
        LL = LeafNode(b, L.parent)

    LL = adjust_tree(tree, L, LL)

    if LL != null:
        // root has split
        old_root = tree.root
        new_root = InternalNode()
        new_root.children.push(InternalNode(calc_bbox(old_root), old_root))
        new_root.children.push(InternalNode(calc_bbox(LL), LL))

        tree.root = new_root

        // fix parent pointers
        for child in tree.root.children:
            child.parent = tree.root
```

B.4 Hilbert tree

```
/// Calculate the Hilbert value of the given integer coordinates.
u64 hilbert_int(u32 x, u32 y):
    u64 h = 0
    for i from 31 to 0:
        x_bit = (x >> i) & 1
        y_bit = (y >> i) & 1
        hh = x_bit << 1 | (y_bit ^ x_bit)
        switch hh:
        case 0:
            // reflect point over the diagonal y=x
            tmp = x
            x = y
            y = tmp
            break
        case 3:
            // reflect point over the diagonal y=U32_MAX-x
            tmp = x
            x = ~y
            y = ~tmp
            break
        h |= hh << (i * 2)
    return h

u32 to_int(f64 d):
    res = d * pow(2, 32)
    if res > U32_MAX:
        // should only happen if d is exactly 1.0
        return U32_MAX
    else:
        return res

/// Calculate the Hilbert value of the given coordinates.
/// x and y must be in the range [0.0, 1.0].
f64 hilbert(f64 x, f64 y):
    u64 h_int = hilbert_int(to_int(x), to_int(y))
    return h_int / pow(2, 64)
```

```
/// Builds a hilbert tree from the bottom up by first sorting
/// the input based on the hilbert values, then merging the nodes
/// in groups of size M to build each level of the tree.
Rtree hilbert_pack(boxes):
    AABB bounds = smallest bounding box that contains all boxes

    vector<(AABB, double)> hboxes
    for box in boxes:
        x, y = rescale point coords to be in the range [0.0, 1.0]
        hboxes.push(box, hilbert(x, y))

    sort hboxes by their hilbert value (second value in the tuples)

    vector<InternalEntry> new_nodes

    // take M hboxes at a time and combine them into a single
    // InternalNode
    while boxes is not empty:
        children = take first (up to) M elements of boxes
        node = LeafNode(children)
        new_nodes.push(InternalEntry(calc_bbox(node), node))

    vector<InternalEntry> old_nodes = new_nodes
    new_nodes.clear()

    while old_nodes.size() > 1:
        while old_nodes is not empty:
            children = take first (up to) M elements of old_nodes
            node = InternalNode(children)
            new_nodes.push(InternalEntry(calc_bbox(node), node))
        old_nodes = new_nodes
        new_nodes.clear()

    return Rtree(old_nodes[0].node)
```

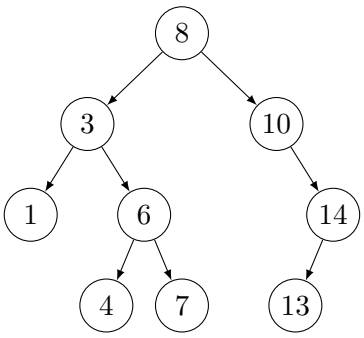
EXAMENSARBETE Comparison of data structures for spatial databases
STUDENTER Alfred Andersson, Hanna Hertzberg
HANDLEDARE Jonas Skeppstedt (LTH)
EXAMINATOR Flavius Gruian (LTH)

Vilket träd flyger bäst?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Alfred Andersson, Hanna Hertzberg**

Ett sökträd är en datastruktur som effektiviserar sökning i stora datamängder, och kan bland annat användas som spatialt index. Detta examensarbete undersöker vilket av de tre sökträden Quadtree, R-tree och Hilbert tree som snabbast kan göra vissa spatiala sökningar, t.ex. att hitta alla punkter inom en viss radie från en given punkt.

På långdistansflyg brukar det finnas en karta som visar planets position, närliggande städer och landmärken, och vilken tidszon planet befinner sig i. För att spara all denna information och sedan snabbt kunna visa upp den på displayen, kan datastrukturen sökträd användas. Detta arbete har jämfört tre vanliga sökträd—Quadtree, R-tree och Hilbert tree—i syfte att finna det vilket som är mest lämpat för att hitta vilken tidszon flygplanet befinner sig i och hitta alla landmärken som ligger inom ett visst avstånd från flygplanet.



Figur 1: Exempel på sökträd där noden som innehåller 8 är roten, noderna med 3, 10, 6 och 14 är interna noder och noderna som innehåller 1, 4, 7 och 13 är lövnoder.

Ett sökträd är en datastruktur som delar upp data på ett sätt som effektiviserar sökning, d.v.s. gör att man snabbare kan hitta önskad information. Sökträd kan se olika ut men alla följer samma grundstruktur, där trädet börjar med en rot som förgrenar sig till interna noder och slutar i lövnoder. När man söker efter t.ex. tidszon använder man flygplanets koordinater för att söka sig igenom trädet från roten ner till den lövnod som innehåller tidszonen flygplanet befinner sig i.

Ett Quadtree delar upp kartan i fyra kvadranter: sydväst, nordväst, sydöst och nordöst. Dessa kvadranter delas sedan upp i mindre subkvadranter baserat på de olika väderstråcken. R-tree delar upp städerna utifrån hur de ligger, d.v.s. noderna innehåller enbart områden där städer finns och inte tom yta. Hilbert tree fungerar liknande som R-tree men sorterar städerna baserat på ett specifikt sätt att ordna koordinater.

Under examensarbetet jämförde vi trädens prestanda, framförallt baserat på hur lång tid det tog att göra sökningar. Vi kom fram till att Hilbert tree var snabbast när det gäller stora datamängder för både tidszoner och städer, och Quadtree är snabbast med mindre datamängder på städer. R-tree var alltid långsammast. I sökandet av städer används mindre datamängder i flygkartan, så i det fallet är Quadtree bäst.

Så, vilket träd flyger bäst? Svår fråga men generellt sett flyger Hilbert tree bäst.