DHBW
Duale Hochschule
Baden-Württemberg

Efficient Queries of Point-Based Geospatial Data - R*Tree vs. HPRTree

Semester Paper

from the Course of Studies Informatik

at the Cooperative State University Baden-Württemberg Heidenheim

by

Josias Raphael Müller

September 2023

Time of Project	9 Weeks
Student ID, Course	4030578, TINF2020AI
Reviewer	Prof. Dr. -Ing. Sabine Berninger

Author's declaration

Hereby I solemnly declare:

1. that this Semester Paper, titled *Efficient Queries of Point-Based Geospatial Data - R*Tree vs. HPRTree* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. I have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Semester Paper has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. I have not published this Semester Paper in the past;

I am aware that a dishonest declaration will entail legal consequences.

Heidenheim, September 2023

Josias Raphael Müller

Abstract

There are many different types of spatial indices, in this work two of them, namely *HPRTree* and *R*Tree* are compared in regards to their size in memory, the time it takes to build an index and the time it takes to delete it again. And how fast they can retrieve elements based on the type of query. For this many different datasets are used - most of which are newly generated for this purpose. In the end, the tests show the *HPRTree* to be superior with regards to all but the query performance where the *R*Tree* outshines.

Contents

Acronyms IV

List of Figures V

Listings VI

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 1
- 1.3 Related Work . . . 2

2 Foundations 3
- 2.1 Relevant Terminology . . . 3
- 2.2 Tools and Libraries . . . 3

3 Approach 5
- 3.1 Variables . . . 5
- 3.2 Metrics . . . 12
- 3.3 Compared Libraries . . . 15
- 3.4 General Considerations . . . 15

4 Evaluation 17
- 4.1 General Remarks . . . 17
- 4.2 Metrics . . . 17

5 Discussion 30

6 Conclusion and Outlook 32
- 6.1 Summary . . . 32
- 6.2 Next Steps . . . 32

Bibliography 33

Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
JVM	Java Virtual Machine
I/O	Input/Output
BBox	Bounding Box
RAM	Random Access Memory
GIS	Geographic Information System
RTree	Rectangle Tree
QTree	Quad Tree
HRTree	Hilbert RTree
HPRTree	Hilbert Packed RTree

List of Figures

3.1 All symmetric datasets. 8
3.2 A selection of the random datasets. 10
3.3 A selection of the randomly clustered datasets. 11
3.4 All real datasets. 12

4.1 Size of all datasets in memory in KiB. 18
4.2 Size of all datasets in memory in percent of theoretical minimum. 19
4.3 Size of the datasets that the *R*Tree* could not handle in memory in percent of theoretical minimum. 20
4.4 Size of the other datasets in memory in percent of theoretical minimum. . . 21
4.5 A selection of build times for element size 12. 22
4.6 A selection of build times for element size 24. 23
4.7 A selection of build times for element size 256. 24
4.8 A selection of build times for element size 512. 25
4.9 A selection of build times for element size 1024. 26
4.10 A selection of deletion times for elements with a size of 24 bytes. 27
4.11 A selection of query all times. 28
4.12 A selection of prepared query times. 29

Listings

3.1 Generation of the coordinates for the symmetric dataset. 7
3.2 Element counts of the synthetic datasets. 9

1 Introduction

1.1 Motivation

The process of preparing data to be quickly retrieved at a later time, usually referred to as *indexing* exists in many forms - one of them is spatial indexing.
Again, there are different concepts behind this term, however here the focus lies on indices for two-dimensional points to later be retrieved through BBox[1] queries.
For example: An index with two points: Berlin, located at 52.518611°, 13.408333° and Ottawa, located at 45.424722°, -75.695°, latitude and longitude respectively - a query for all points contained within 50°-54°, 12°-15° would only return Berlin because Ottawa is outside of the specified area.
The motivation for this work lies in comparing libraries that provide this functionality in regards to performance as to determine which is best under what circumstances and how to adjust circumstances to improve performance where possible.

The entire source code of this work is available under `https://github.com/josias-mueller/letztestudienarbeit`.

1.2 Objectives

The following objectives should be met for all libraries and data structures taken into account:

1. Determine the required size in memory under different circumstances, and especially how it scales with respect to element count and size

   Barring compression, it is fairly easy to determine how much size is necessary to save a set of elements in memory - *element count* * *element size*. Any memory above that is overhead required for the index itself. The lower the required memory is the more elements fit into memory.

2. Determine the time to build the index under different circumstances, and especially how it scales with respect to element count and size

[1] The term Bounding Box in this work refers to an axis-aligned rectangle used to specify an area of interest.

Build time refers to the time it takes to go from a set of elements to an index structure that is ready to be queried.

3. Determine the time to delete the index under different circumstances, and especially how it scales with respect to element count and size.

   Delete time refers to the time it takes for the environment to clean up the index, this is mostly freeing the allocated memory in this case.

3. Determine the time to carry out queries under different circumstances, and especially how the query performance scales with the number of elements in the index, their size and how many elements are included within the query region.

   As the primary and most frequent operation the query performance is by far the most important metric, as the frequency of this operation is so high comparably it may very well be time-efficient to accept longer build times to achieve better query performance.

1.3 Related Work

A roughly related work[1] introduces the HRTree as an improved RTree / *R*Tree*.

Another roughly related work[2] presents *R*Trees* conceptually and carries out benchmarks that suggest, that *R*Trees* are generally better than QTrees and *Greene's RTrees*.

A different related work[3] compares different packing algorithms for RTrees - also making references to Hilbert curve based packing used in HPRTrees. It concludes that none of the considered packing methods is optimal for all datasets.

There is another related work[4] which compares QTree and RTree index implementations for the spatial Oracle database. In the used benchmarks the RTrees consistently outperform the QTrees.

PostGIS, a *Postgres* database extension, enabling it do perform GIS-operations has documentation[5] on the internally used spatial index (this is also an RTree).

A further related work[6] is an online resource providing an overview regarding different spatial indexing techniques, their strengths and weaknesses. This one does not have a reference to HRTrees or HPRTrees though.

2 Foundations

2.1 Relevant Terminology

Unless otherwise noted these definitions are used:

***Float* refers to a 4 byte floating point number type as defined in IEEE 754-2008 (*binary32*).**

***Point* refers to an object consisting of two floats, representing two coordinates in some form of coordinate system.**

***Element* refers to an object consisting of a point plus optional data.**

***Tree* refers to a hierarchical data structure composed of connected nodes.**

2.2 Tools and Libraries

The following is a selection of the most useful tools and libraries that are used in this work.

2.2.1 Leaflet

Leaflet[7] is a JavaScript library that can be used to create maps. In this work it is used to create visualisations for the different datasets.

2.2.2 GeoJSON

GeoJSON[8] is a JSON[9] based format for coordinate based data. In this work it is used to format some data to generate visualisations with leaflet (see subsection 2.2.1).

2.2.3 Rust

Rust[10] is a programming language that enables low level control of a program while still being ergonomic to write. In this work it is used for the main benchmarking as it provides great performance and sufficient control over (de)allocation and general execution to adequately measure it.

2.2.4 Python

Python[11] is useful because of its plethora of very useful libraries. In this work it is utilised for some of the data generation and for all visualisations, barring the ones generated with *Leaflet* (see subsection 2.2.1).

2.2.5 Matplotlib

Matplotlib[12] is the most used Python library in this work. It is used to generate all of the data plots.

3 Approach

This chapter describes how the benchmarking was approached in this work. It also describes what data was generated how to facilitate the benchmarks in a repeatable manner.

3.1 Variables

The concept of benchmarking rests on determinism: The expectation that if none of the input variables of a process change, the output remains the same - no matter how many times it is executed. If this is not the case, that means that one or more variables have been overlooked. The intention is to learn all variables relevant to the output of a process, learn how exactly they modify the output as to be able to manipulate them to create a favorable change in the output.

During benchmarking all variations of a predetermined set of each of the following variables are tested, except for those that cannot be executed, e.g. because of memory constraints:

3.1.1 Element Size

Varying the size of the indexed elements themselves primarily affects the memory consumption. If the size of the elements is increased, this increases the theoretical minimum of memory needed to store the data. However, since transferring more data takes more time the expectation is that everything will slow down as the element size increases. In addition to this, increasing the size per element also means that fewer elements fit in a *cache*[1] *line*[2].

There are 5 different element sizes used in the benchmarks:

- Element: 12 Bytes (latitude and longitude as floats and a 32 bit id)
- BiggerElement: 24 Bytes (latitude and longitude as floats and 128 bits of data)
- BigElement: 256 Bytes (latitude and longitude as floats and 31 64 bit data fields)

[1] As memory size increases, it becomes slower - cache refers to a secondary part of memory, that is smaller and thus faster.

[2] A cache line is the smallest unit of memory that can be mapped to the CPUs's internal cache

- VeryBigElement: 512 Bytes (latitude and longitude as floats and 63 64 bit data fields)
- VeryVeryBigElement: 1024 Bytes (latitude and longitude as floats and 127 64 bit data fields)

3.1.2 Element Count

Varying the number of elements in the index has a similar effect as varying the size in that the theoretical minimum of memory needed to store the data is increased. However, the number of elements that fit in a cache line stays the same. The smallest used dataset consists of 16,200 elements where the largest contains 4,147,200 elements. The expectation is that everything will slow down as more and more elements are added to the index.

3.1.3 Element Distribution

Varying element distribution is not quite as simple as varying the element size or the number of elements as it is not just a number. Inspecting how the indices behave with different distributions is interesting as this information could be used to pick the best index based on the distribution of the data - should the results prove significant.

3.1.4 Datasets

In abstraction, every dataset could be viewed as a combination of two additional variables: element count and element distribution.

Data Origins

Real Real data, as in data points extracted from physical reality are interesting because they are the most obvious application for indices as described in this work.

Synthetic Synthetic data is convenient as it can just be generated. However, a system under synthetic load (or using synthetic data in general) may very well not behave the same way it would under a real load (or with real data respectively).

To create a better understanding of the used datasets there will be visualisations showing every datapoint on a map, this visualisation is created by serialising all of the datapoints in a dataset as GeoJSON (refer to subsection 2.2.2) and loading all of this data with Leaflet (refer to subsection 2.2.1). In addition to this, to provide a frame of reference, in all of these visualisations a GeoJSON dataset[13] that outlines all countries on earth is displayed beneath the actual dataset.

Synthetic - Symmetric

The generated *Symmetric* dataset is constructed as follows:

```
1    for every multiplier in [1, 4, 16, 64, 256]
2        generate all combinations where
3            x ∈ [-180; 180), step = 2/√multiplier
4            and
5            y ∈ [-90; 90), step = 2/√multiplier
```

Listing 3.1: Generation of the coordinates for the symmetric dataset.

This results in $multiplier * 180 * 90$ points per dataset:

Multiplier	1	4	16	64	256
Number of elements	16,200	64,800	259,200	1,036,800	4,147,200

Table 3.1: Number of elements in each symmertic dataset.

When drawn on a map these datasets may look as shown in Figure 3.1[1].

[1] Fun fact: Figure 3.1e took Leaflet about half an hour and 80 GB of memory to render - that was surely worth it, right?

(a) Symmetric dataset with multiplier 1.

(b) Symmetric dataset with multiplier 4.

(c) Symmetric dataset with multiplier 16.

(d) Symmetric dataset with multiplier 64.

(e) Symmetric dataset with multiplier 256.

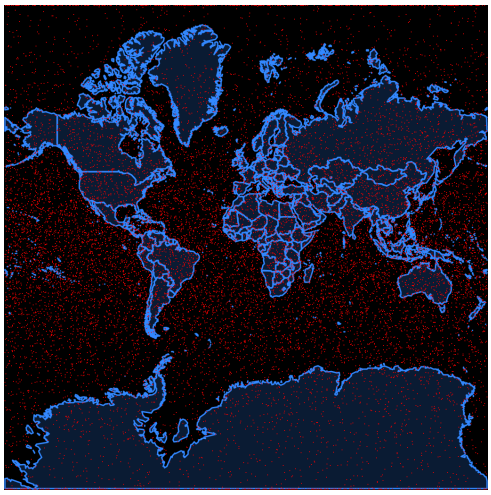
Figure 3.1: All symmetric datasets.

Synthetic - Random

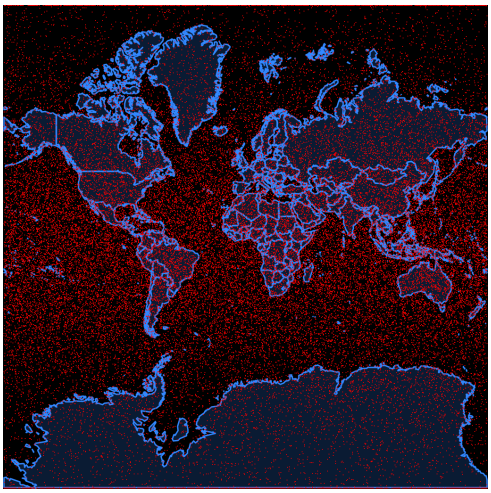
This dataset is constructed completely randomly via a uniform distribution - no limitations besides the ones already defined in Listing 3.1. Since the intention with this dataset is to determine whether element distribution affects performance, the other variable - in this case element count should be eliminated. To achieve this, the number of elements in this dataset is derived from the element count of the other datasets. To reduce the chances of accidentally generating a dataset that favours one implementation over the other, this dataset has been generated and tested multiple times for each element count. The expectation with this is that there are no significant differences in the results of the datasets generated with the same parameters. The complete list of element counts is as follows in Listing 3.2:

```
1    16,200, 44,692, 64,800, 140,974, 259,200, 1,036,800, 3,808,651,
         4,147,200
```

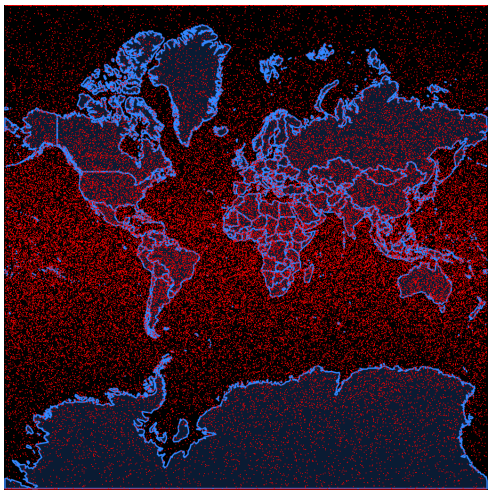
Listing 3.2: Element counts of the synthetic datasets.



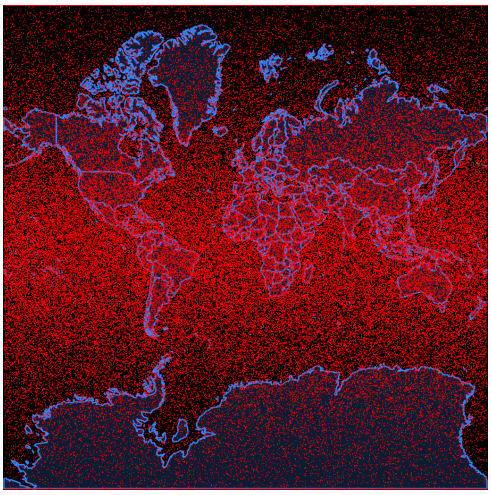
(a) The first of three random datasets with an element count of 16200.



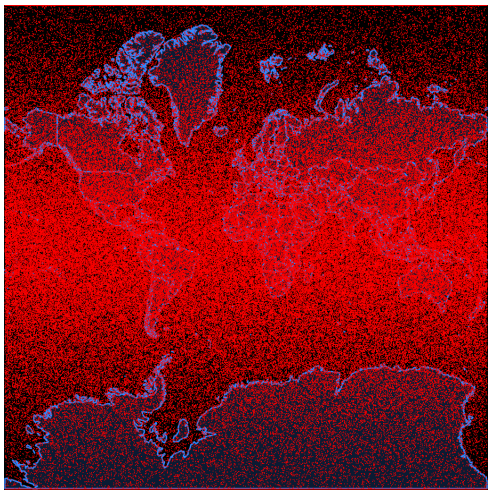
(b) The first of three purely random datasets with an element count of 44692.



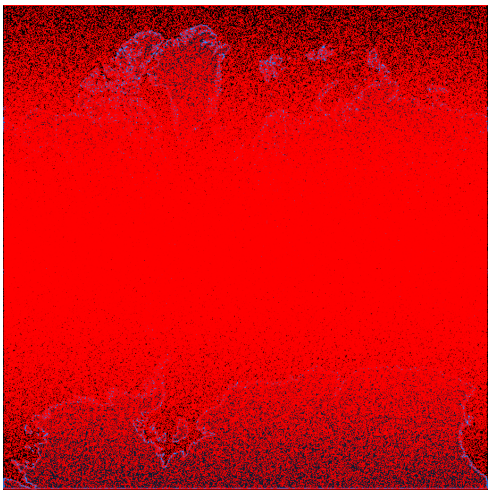
(c) The first of three purely random datasets with an element count of 64800.



(d) The first of three purely random datasets with an element count of 140974.



(e) The first of three purely random datasets with an element count of 259200.

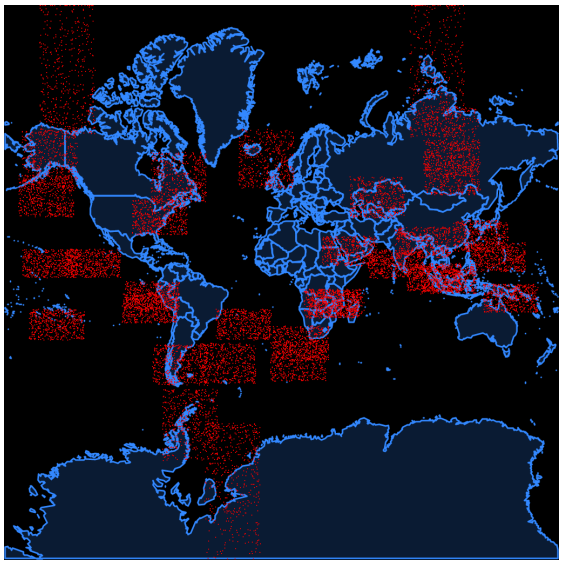


(f) The first of three purely random datasets with an element count of 1036800.

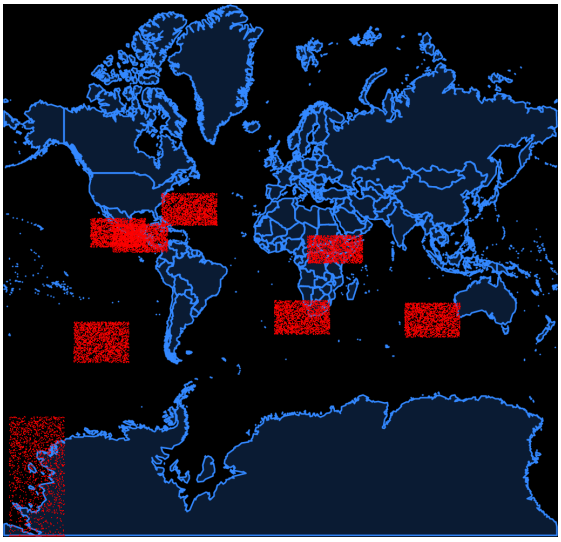
Figure 3.2: A selection of the random datasets.

Synthetic - Randomly Clustered

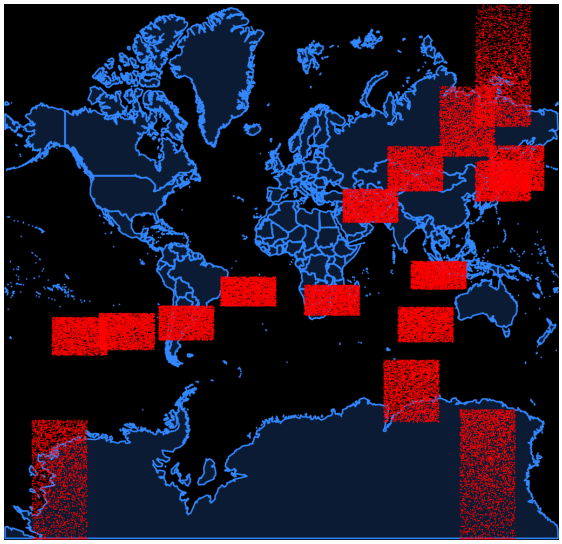
The clustered dataset is constructed like the random dataset in subsubsection 3.1.4, except that the points are generated in a radius around randomly generated points as to create distributed clusters of points. This is done to further test how distribution affects performance. This is closer to how elements are distributed in a real dataset when compared to a fully uniform distribution while still being easy to generate.



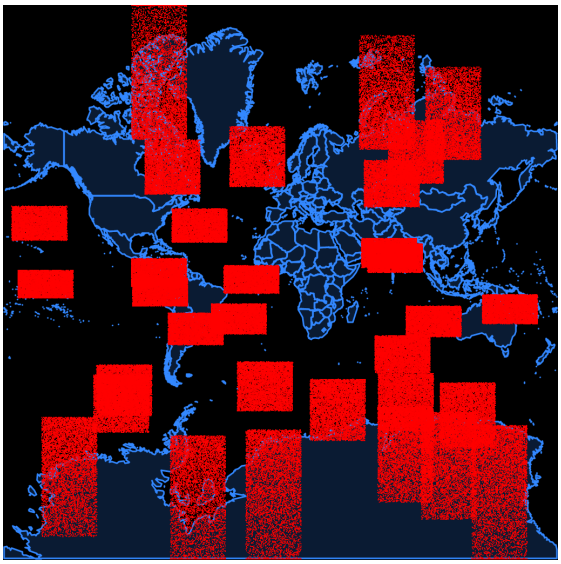
(a) The first of three randomly clustered datasets to have 506 elements in 32 clusters.



(b) The first of three randomly clustered datasets to have 2025 elements in 8 clusters.



(c) The third of three randomly clustered datasets to have 4050 elements in 16 clusters.

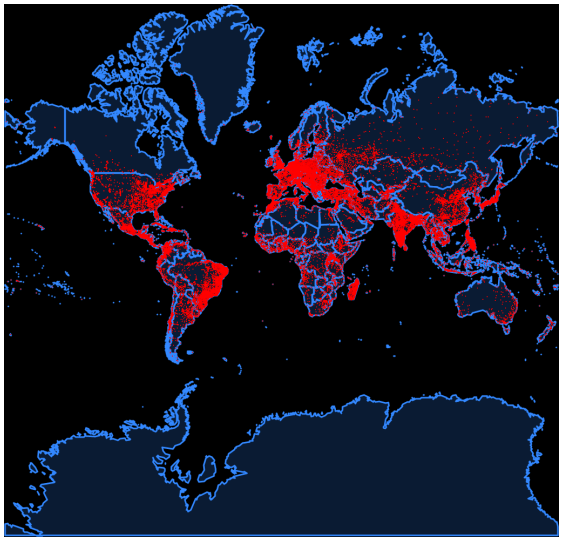


(d) The third of three randomly clustered datasets to have 8100 elements in 32 clusters.

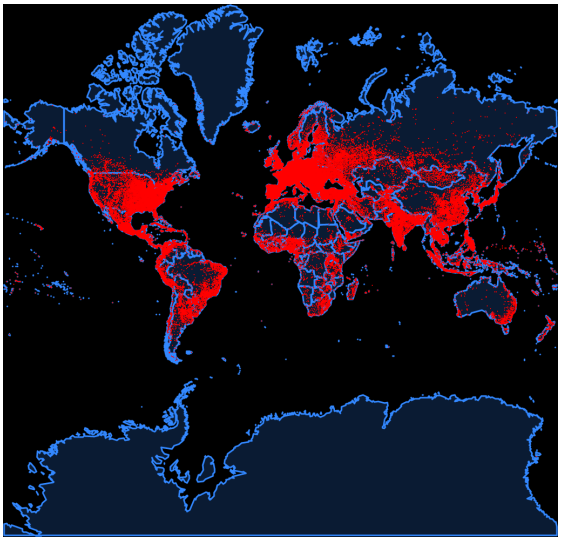
Figure 3.3: A selection of the randomly clustered datasets.

Real

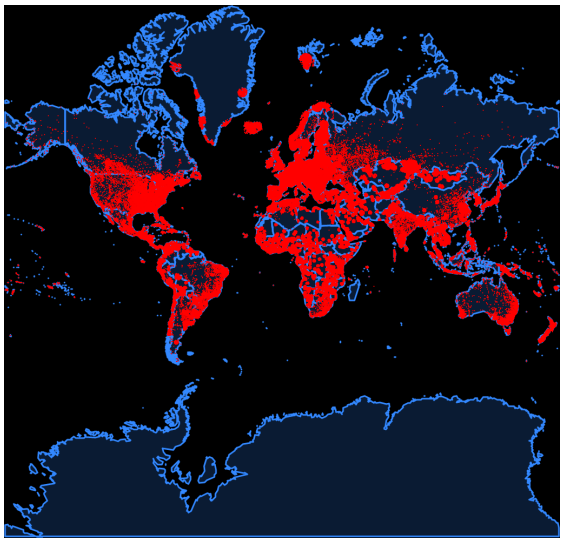
There are multiple different real datasets in use: The first[14] and smallest consists of 44,692 points representing prominent cities. The second[15] contains 140,974 cities and the third[16] and final dataset incudes 4,384,909 points in theory - in practice, with invalid entries (such as missing coordinates) excluded, it still includes 3,808,651 entries.



(a) The first of three real datasets (44,692 elements).



(b) The second of three real datasets (140,974 elements).



(c) The third of three real datasets (4,384,909 elements).

Figure 3.4: All real datasets.

3.2 Metrics

Multiple metrics are used to compare the different datasets:

3.2.1 Size in Memory

While computers today do have a lot of memory by comparison, the memory requirements for nearly everything have grown quite a bit as well and so have the available / necessary datasets. In addition to this, the smaller something is in memory, the faster it can be iterated through - a smaller memory footprint is always an advantage.
This metric measures the sum of all allocations for each data structure. It does not take memory layout into account. It also does not count over-allocation[1]. So a single allocation of 1MiB would count the same as 1024 allocations that request 1 byte each, even though (in reality) the latter would most likely use up more memory and cause heap fragmentation.

3.2.2 Time to Build

Being able to build a data structure faster enables a higher update frequency for internally immutable data structures. A higher update frequency means more current information being exposed through the index. When optimising for read performance, using internally immutable data structures makes sense as this removes the need for synchronisation, fencing and similar mechanisms (which hurt performance to be able to guarantee deterministic / safe behaviour) when utilising parallelisation and/or asynchronicity. The latter, again, just makes sense when optimising for performance - which is probably why one of these indices would be used in the first place.
This metric measures the time it takes for the structure to load the data and to modify that data so that efficient querying is possible. To improve comparability, the dataset is always preloaded into memory first - which is not measured. This way I/O performance (which can be quite flakey) is cut out of the equation. To make sure that all of the data was actually loaded, the number of elements in the index is compared with the number of entries in the source dataset - this is also not measured.

3.2.3 Time to Delete

Being able to delete a data structure faster means that the previously allocated memory is made available to be allocated again more quickly. This can help to keep the average memory consumption of a process down. Deletions, or rather deallocations, are constant-time actions, generally. However, if a data structure allocates multiple times, even more so if it does so recursively, it also needs to deallocate multiple times - so the more allocations,

[1] While allocator APIs generally allow to request an arbitrary number of bytes, most do not guarantee that the returned memory block is exactly the requested size - only that it is at least as big. This can be for several reasons: To work against fragmentation, to store additional information about the allocation or to make future in-place growing of the allocation possible.

the more deallocations, the longer deletion takes. This is even worse if pointer-chasing[1] is necessary, like with linked lists, or even more so with tree-like structures where a node owns multiple other nodes recursively.
To get this metric, the previously built data structure is manually deallocated - the execution time of this deallocation function is measured.

3.2.4 Time to Query

Querying is the core functionality of indices, all preparation beforehand is done so that querying can be as efficient as possible. Faster queries simply means that more queries can be done per second - so to achieve a higher throughput (unless there is a bottleneck somewhere else - which in a networked environment, like with a server, is quite likely).
This metric is not as straight forward to measure as the others, there are infinitely many possible queries for each dataset (if we had infinite precision arithmetic that is). So there is only one option: Picking a subset of queries that share significant parameters. For this purpose two kinds of queries will be used:

Query All

For the *query all* benchmark, a given index is simply asked to return all points in its extent. This has one obvious flaw: It is dependent on the size of the index. When comparing an index with one that is twice as big, the latter has to do at the very least twice as much work just moving the data. So comparing this is mostly useful for indices with the same raw amount of data. Otherwise the resulting number could potentially be given relative to the ratio between the sizes of the two datasets.

Prepared Queries

The idea behind the prepared queries is as follows: It should be possible to randomly generate a set of BBoxes for every dataset so that each one contains n elements. This is then done 16 times for every $n \in \{16, 64, 256, 1024, 4096\}$. Now it is possible to go through every dataset and measure how long it takes to retrieve n elements and analyse, how the element size and count affect this and how well the different indices handle it for each of the element distributions.

[1] When data can only be reached through a series of pointers, where one points to the next. This is generally necessary for dynamically sized data structures that are not stored in a contiguous block of memory - through to different degrees, based on the data structure.

3.3 Compared Libraries

At this stage two libraries are included in this comparison:

rstar The first library implements an *R*Tree*, it is found on *Rust's* package index under the following url: `https://crates.io/crates/rstar` at version *0.11.0*.

hprtree The other library implements an HPRTree, it is found on *Rust's* package index under the following url: `https://crates.io/crates/hprtree` at commit *df396bd7090a8370-78aafac2b5a0cec575872df6* of the master branch (one commit after version *0.2.2*).

3.4 General Considerations

A few general considerations are necessary before the benchmarking can commence:

Priority As processes running on multitasking operating systems share (and thus compete) for the same resources, they affect each others performance. Because of this, the process priority of the benchmarking process is set to the maximum available (24 - *Real Time*) on the Windows operating system. While I/O-operations are intentionally excluded from the benchmarks having a lower I/O priority should not matter. Still this one is also set to the maximum available (in this case *High*).

Swapping Another variable that can affect performance is swapping (also called paging): Computers tend to be allowed to move RAM content to disk. This is great because it provides the computer a lot of capacity, however this is also bad because this can cause hard (sometimes also called major) page faults. This happens if a program attempts to access a section of memory that is not currently located in RAM - when this happens the operating system has to load that piece of memory in the the resulting fault handler, which results in a lot of (potentially inconsistent) latency as the memory access has essentially become an I/O-operation. Sometimes the operating system might also have to page another section of memory out, before it can page the requested section in, which makes the aforementioned effects even worse. For these reasons, swapping is disabled on the machine carrying out the benchmarks. This also means that if a datastructure attempts to allocate more memory than is physically available, it will fail to allocate - in many environments this is an unrecoverable error (the only exceptions the author can think of are the kernel finding and killing other processes that use up sufficient memory in the

fault handler or a memory-managed environment like the JVM attempting to compact the heap).

4 Evaluation

In this chapter, the results of the benchmarks are presented.

4.1 General Remarks

The benchmarks concluded after 44 hours of continuous runtime, resulting in about 5.4GiB of data.

For visual consistency, the colours will remain the same for all visualisations: Green elements reflect the behaviour of the *HPRTrees* and orange elements reflect that of the *R*Trees*. All values are rounded - how much depends on the value range but values that are being compared with one another are always rounded the same way.

4.2 Metrics

The benchmarks collected data on the following metrics:

4.2.1 Size in Memory

Several interesting observations can be made in relation to the indices size in memory. Figure 4.1 shows the size in memory of all datasets together with the theoretical minimum. In this graph it looks like the HPRTree was smaller for all datasets. The *R*Tree* even failed to build for several datasets - this is visualised through datapoints below the theoretical minimum. The main problem with this graph is that due to the wide range of dataset sizes, the lower end is near impossible to read.

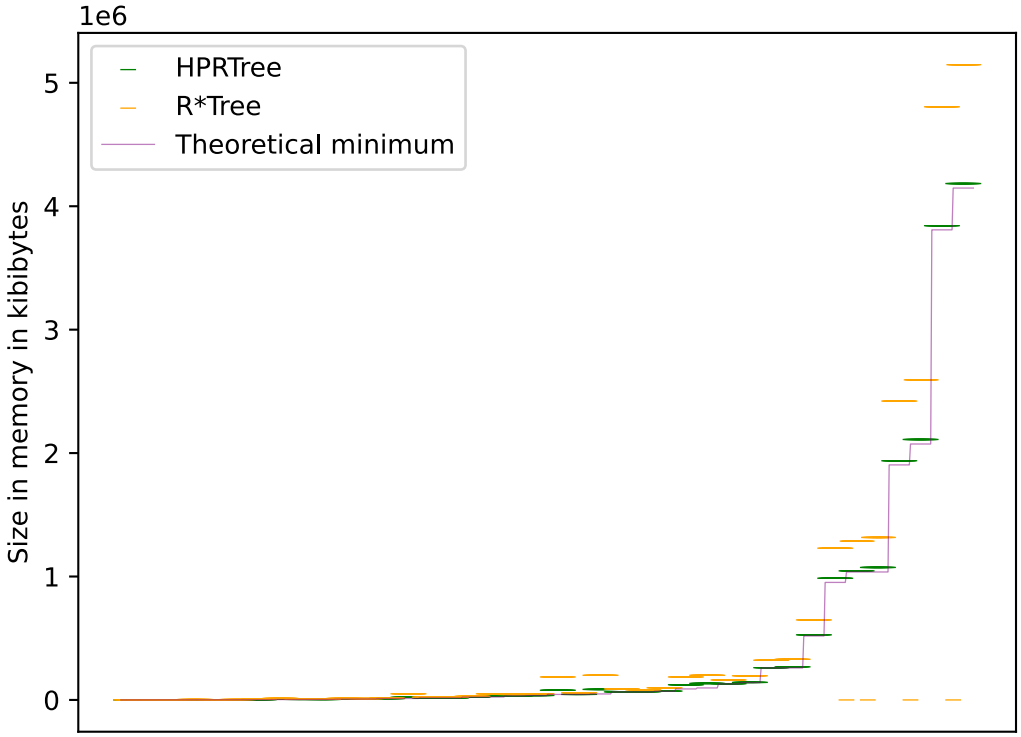


Figure 4.1: Size of all datasets in memory in KiB. If a datapoint is below the theoretical minimum, that means that it ran out of memory.

Figure 4.2 improves on this by presenting all values in percent of the theoretical minimum. This corroborates the previously made claims: The HPRTree is always faster than its counterpart.

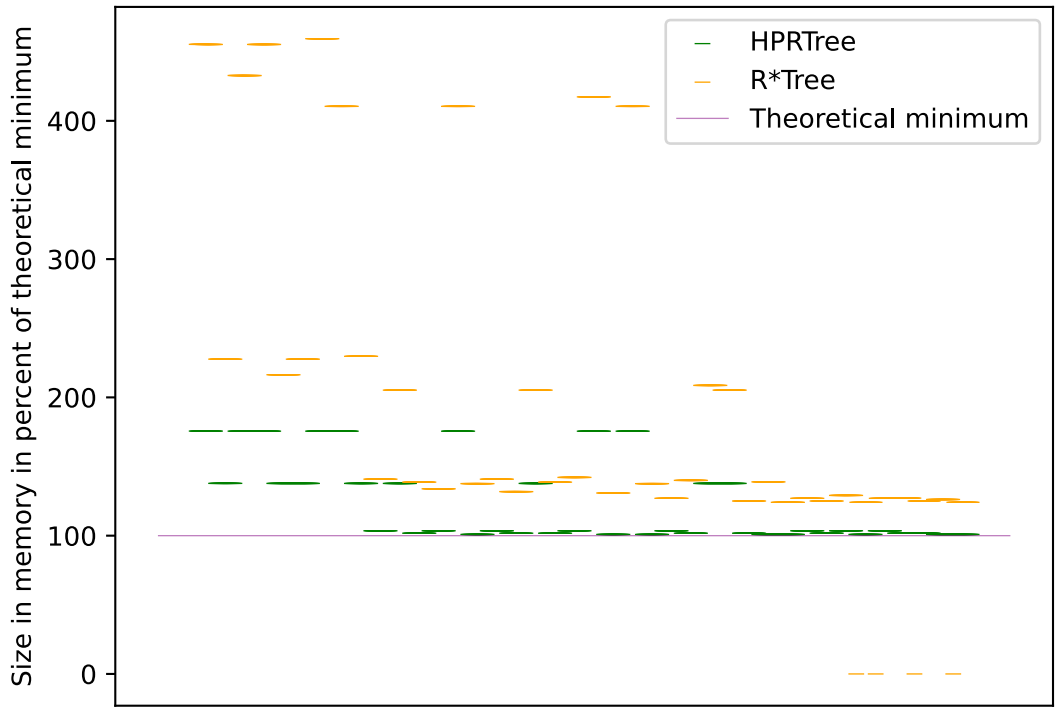


Figure 4.2: Size of all datasets in memory in percent of theoretical minimum. If a datapoint is below the theoretical minimum, that means that it ran out of memory.

Looking at the numeric values, this can be concretised a bit more: The best efficiency[1] the HPRTree reached was circa 100.9 percent of the theoretical minimum, at worst it was circa 175.6 percent. The *R*Tree* managed 124.1 percent at best and 459.4 percent at worst. In direct dataset-by-dataset comparison the HPRTree required 38.2 percent of what the *R*Tree* needed at best and 81.5 percent at worst.

Figure 4.3 takes a closer look at the datasets that the *R*Tree* could not handle with the given resources. Here it becomes clear that while the *R*Tree* failed, the HPRTree was near peak efficiency - this is probably because these are some of the biggest datasets, namely the synthetic datasets with a multiplier of 256 and element sizes of 256, 512 and 1024 and the synthetic dataset with a multiplier of 64 and an element size of 1024. Because the size requirement for the additional data scales logarithmically for the HPRTree, the overhead becomes less noticably as the theoretical minimum grows.

[1] As the values are in percent of the theoretical minimum, the best possible efficiency is 100%, the higher the value the worse the overhead.

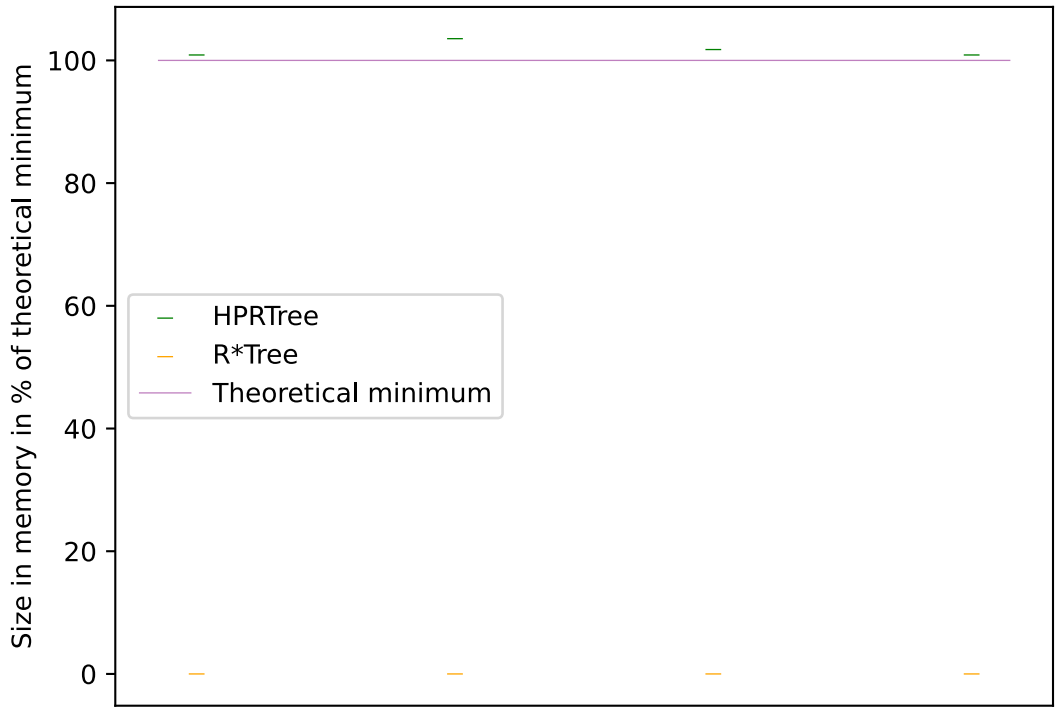
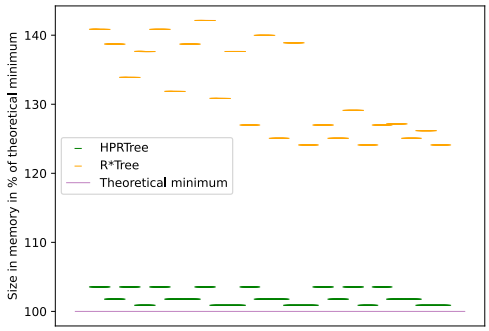
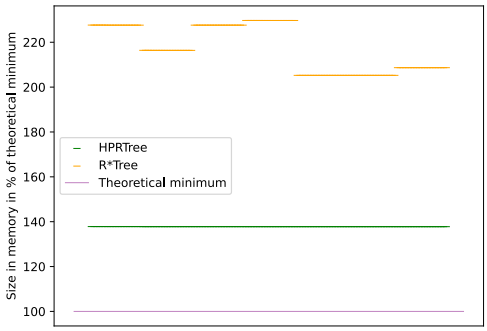


Figure 4.3: Size of the datasets that the *R*Tree* could not handle in memory in percent of theoretical minimum. If a datapoint is below the theoretical minimum, that means that it ran out of memory.

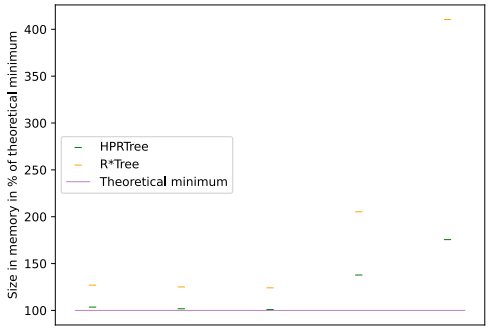
Figure 4.4 shows the rest of the datasets grouped by value ranges.



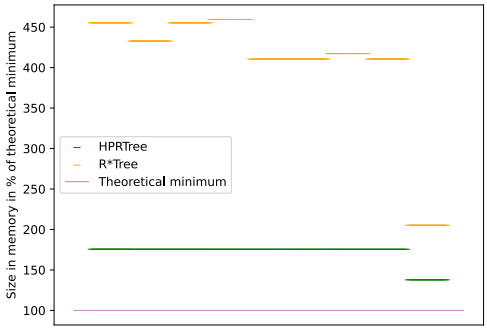
(a) Datasets between 100 and 140 percent of the theoretical minimum.



(b) Datasets between 140 and 220 percent of the theoretical minimum.



(c) Datasets between 100 and 400 percent of the theoretical minimum.



(d) Datasets between 140 and 450 percent of the theoretical minimum.

Figure 4.4: Size of the other datasets in memory in percent of theoretical minimum.

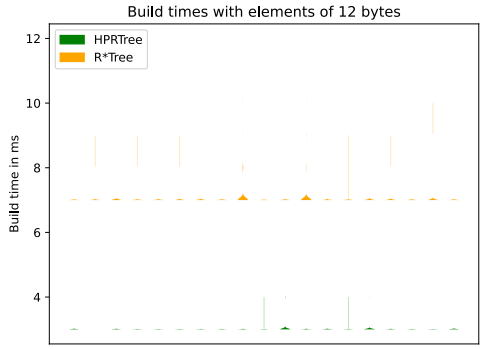
4.2.2 Build Time

The data for build time looks similar: When compared to the build time of the *R*Tree* the HPRTree is at worst 1.8 times faster and at best 11.7 times faster. Again, there is not a single dataset where the *R*Tree* beats the HPRTree. This appears to be strongly correlated with the element size:

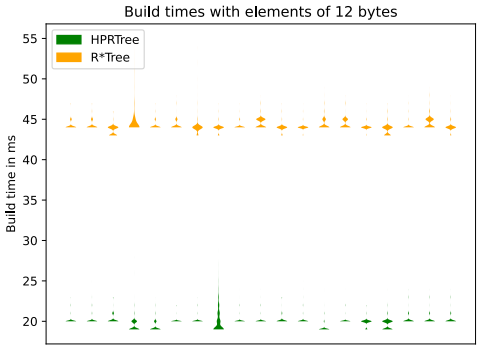
Element size	Difference at worst	Difference at best
12	1.8	2.8
24	1.9	3.0
256	3.3	5.3
512	4.1	11.7
1024	5.4	10.2

Table 4.1: Best and worst case factors for the build times. Higher is better for the HPRTree.

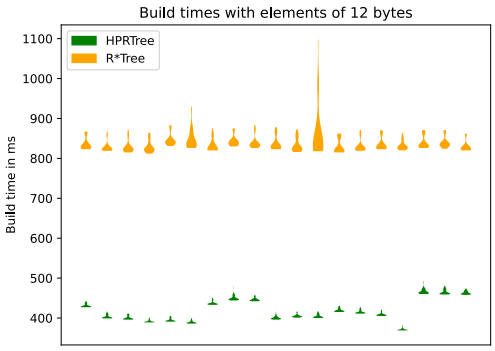
The only reason why the difference at best is lower for the biggest element, is persumably that the datasets that would be even worse for the *R*Tree* do not fit into memory. Figure 4.5 shows plots for a selection of datasets with an element size of 12, they provide more detail than the range given in Table 4.1.



(a) Build times for element counts between 16200 and 44692.



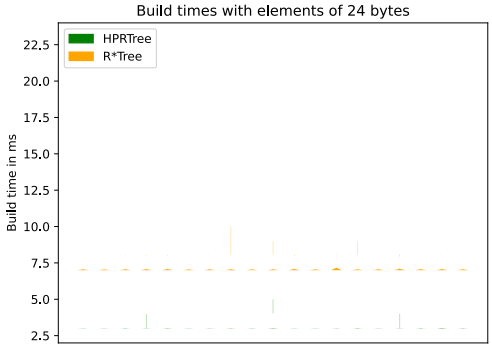
(b) Build times for element counts between 140974 and 259200.



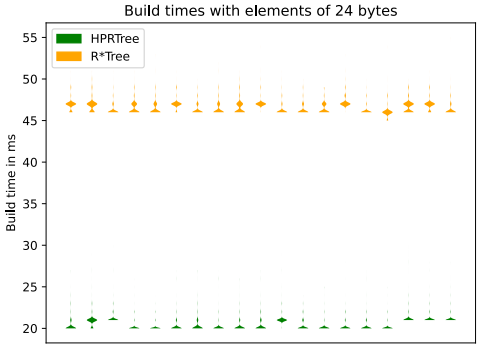
(c) Build times for element counts between 3808651 and 4147200.

Figure 4.5: A selection of build times for element size 12.

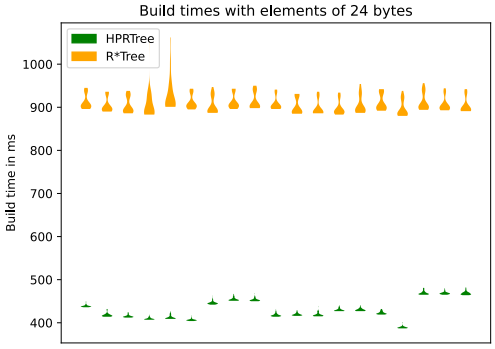
Figure 4.6 shows plots for a selection of datasets with an element size of 24.



(a) Build times for element counts between 16200 and 44692.



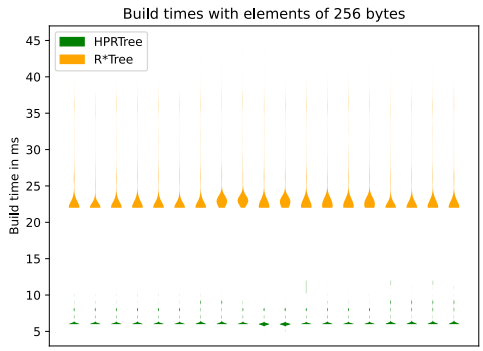
(b) Build times for element counts between 140974 and 259200.



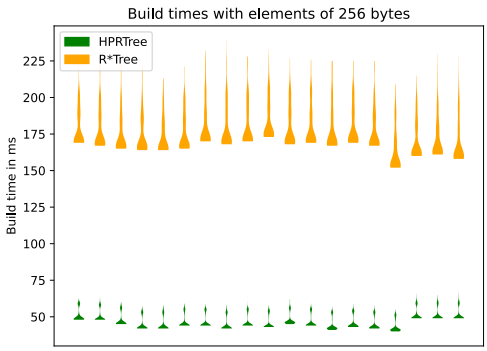
(c) Build times for element counts between 3808651 and 4147200.

Figure 4.6: A selection of build times for element size 24.

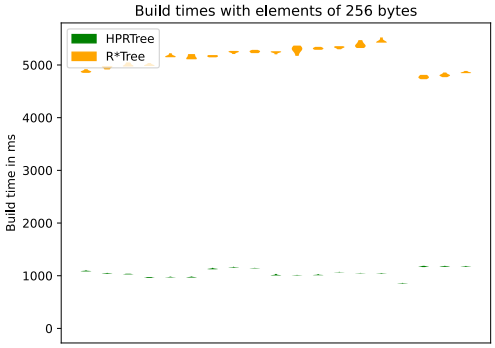
Figure 4.7 shows plots for a selection of datasets with an element size of 256.



(a) Build times for element counts between 16200 and 44692.



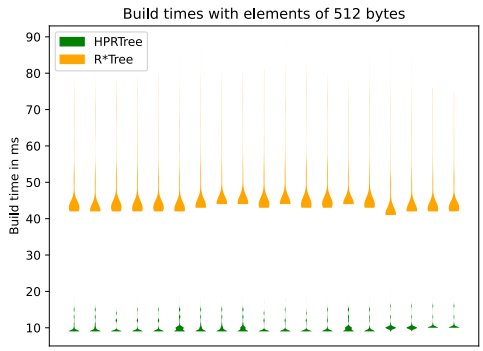
(b) Build times for element counts between 140974 and 259200.



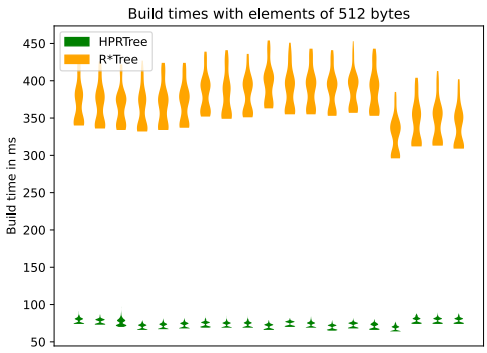
(c) Build times for element counts between 3808651 and 4147200.

Figure 4.7: A selection of build times for element size 256.

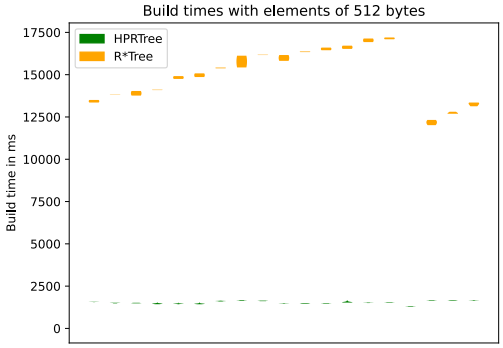
Figure 4.8 shows plots for a selection of datasets with an element size of 512.



(a) Build times for element counts between 16200 and 44692.



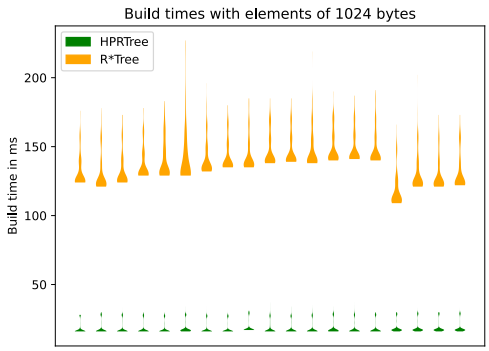
(b) Build times for element counts between 140974 and 259200.



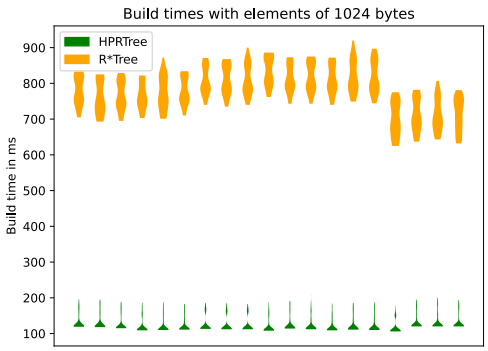
(c) Build times for element counts between 3808651 and 4147200.

Figure 4.8: A selection of build times for element size 512.

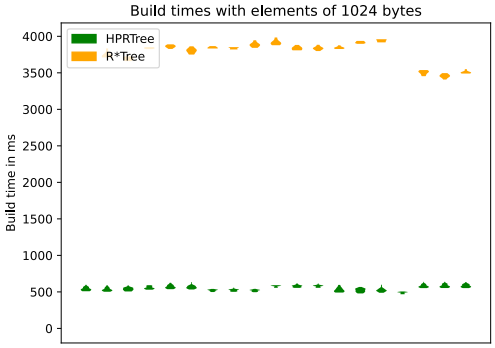
And finally, Figure 4.9 shows plots for a selection of datasets with an element size of 1024.



(a) Build times for element counts between 16200 and 44692.



(b) Build times for element counts between 140974 and 259200.



(c) Build times for element counts between 259200 and 1036800.

Figure 4.9: A selection of build times for element size 1024.

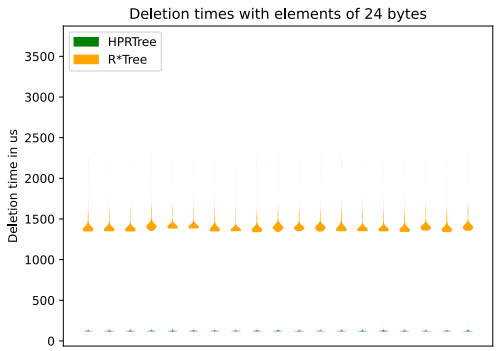
4.2.3 Deletion Time

As deletion time is fairly tightly coupled with build time (or at least the allocation portion of it), it is no big suprise that the results here are similar to the build times'. The HPRTree is better in every benchmark. It is always at least 1.9 times faster but this can go up to 72.9 times faster. However, different to how the build time behaved, these times seem to be more strongly correlated with the element count and not so much with the size of the individual elements as is shown in Table 4.2:

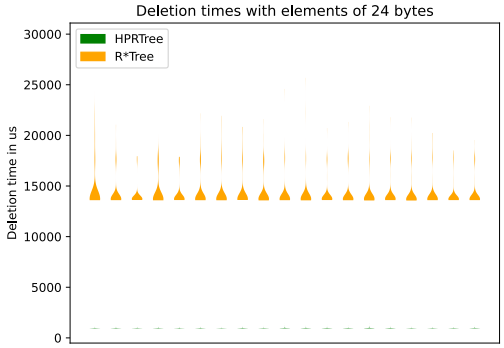
Element count range	Difference at worst	Difference at best
(16,200;44,692]	1.9	6.9
(44,692;64,800]	2.6	13.8
(64,800;140,974]	6.7	17.6
(140,974;259,200]	5.6	15.0
(259,200;1,036,800]	7.6	34.8
(1,036,800;3,808,651]	9.6	68.0
(3,808,651;4,147,200]	11.9	72.9

Table 4.2: Best and worst case factors for the deletion times. Higher is better for the HPRTree.

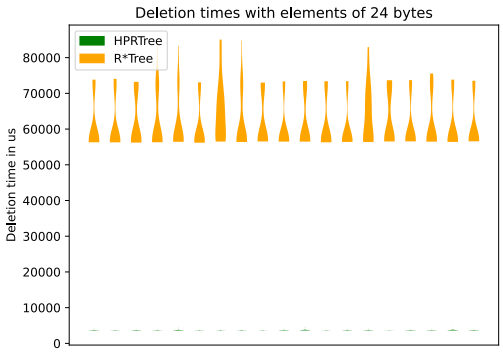
Figure 4.10 shows plots for a selection of datasets with an element size of 24.



(a) Deletion times for element counts between 64800 and 140974.



(b) Deletion times for element counts between 259200 and 1036800.

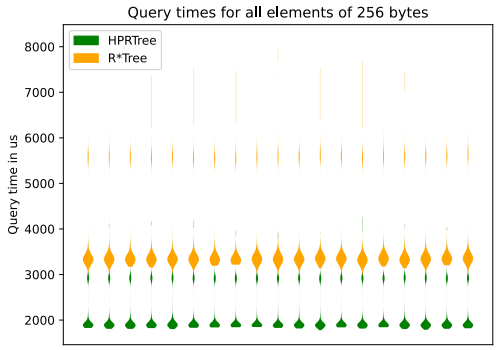


(c) Deletion times for element counts between 3808651 and 4147200.

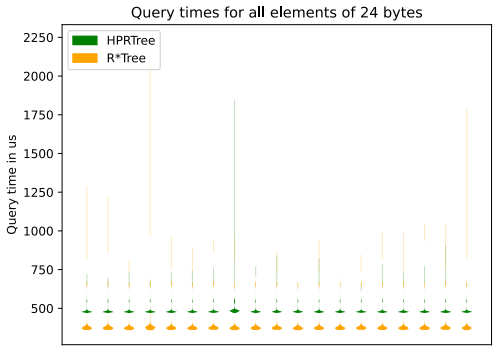
Figure 4.10: A selection of deletion times for elements with a size of 24 bytes.

4.2.4 Query All Time

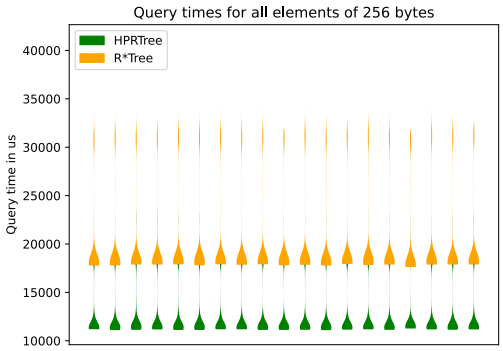
Retrieving all elements from the indices is the first metric where HPRTree does not win everything across the board. It is still ahead in 91% of datasets, but in the worst case it is 20% slower than the *R*Tree*. In the best case it is still about 3.2 times faster than the *R*Tree*. The HPRTree beats the *R*Tree* in all datasets with element size 12, 256, 512, all but one with element size 1024 and in about half of element size 24. There appears to be a slight trend where the greater the element count gets, the better the HPRTree performs, and the reverse seems to be true with element size. The smaller each element gets the better the HPRTree appears to perform. The trend makes sense, however what is quite odd is how the *R*Tree* just wins for some datasets without a trend that would be immediately obvious. A selection of these datasets can be seen in Figure 4.11.



(a) Query all times for element counts between 16200 and 44692.



(b) Query all times for element counts between 16200 and 44692.

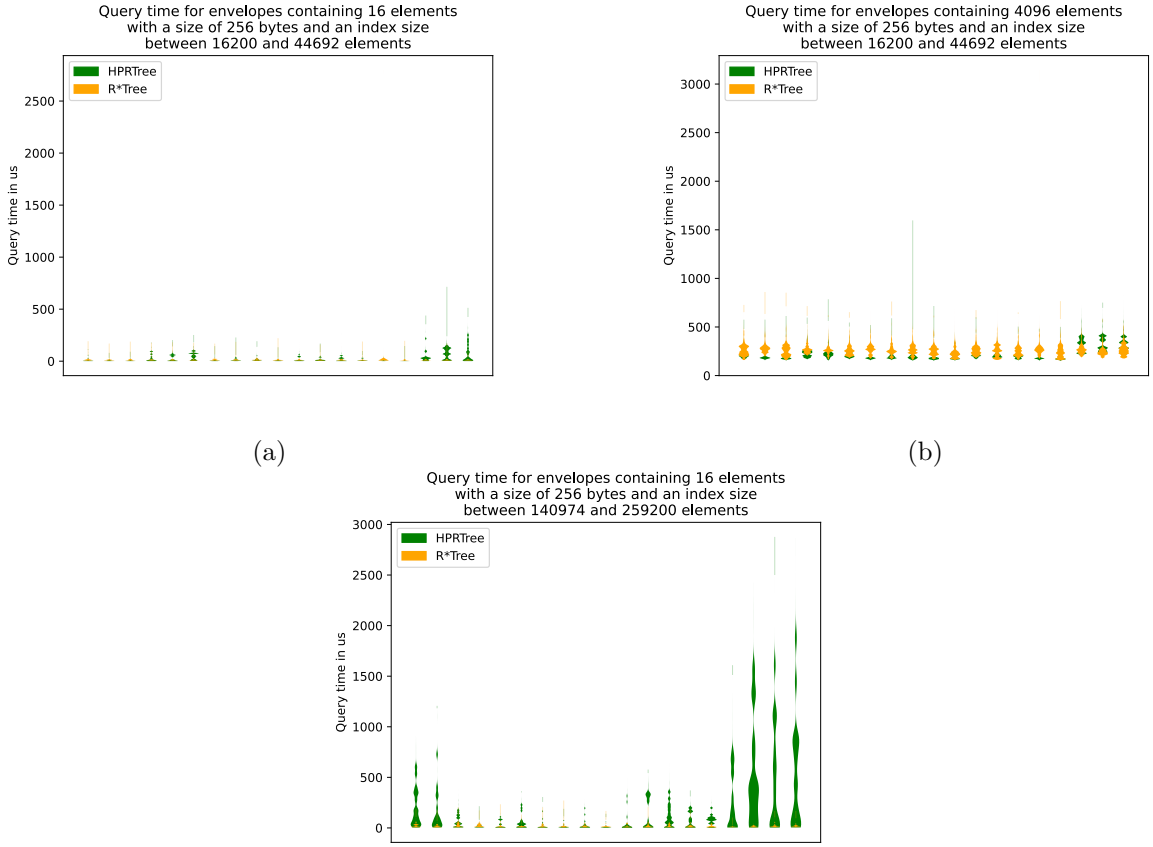


(c) Query all times for element counts between 140974 and 259200.

Figure 4.11: A selection of query all times.

4.2.5 Prepared Query Time

Overall, the HPRTree is slower in about 70% of datasets and sometimes by quite a bit too - at worst the HPRTree is about 99.8% slower than the *R*Tree*. In this case that is 26.6ms versus 0.06ms - this is for the second fully random dataset with 3,808,651 elements and an element size of 1024. Even at best the HPRTree is *only* about 4.3 times faster than the *R*Tree*. The only datasets where the HPRTree consistently outperforms its competitor here are the first[14] and second[15] real datasets - the third dataset is about even between the two.



(a) (b)

(c)

Figure 4.12: A selection of prepared query times.

5 Discussion

The benchmarks were carried out on a Windows computer with 64GB of 3200MHz DDR4 Memory on two channels and a Ryzen 7 3700X (8x4.2GHz) Processor with no other resource intensive programs running. The presented numbers will vary depending on the hardware they are generated on, but the expectation is that on similar hardware at the very least the relation between the numbers remains similar.

Due to time constraints, the benchmarks were limited in execution time. This has resulted in a low sample size for the slower operations. This is problematic for determining statistical significance, but considering how far apart the values are and how tight the spread is this should not matter.

Another limitation due to time constraints is the selection of libraries and indices, of which there are only two. However, the two that are included provide a good starting point: The *R*Tree* as the default choice for this type of work and the HPRTree as an optimisation promising near perfect space-utilisation and good locality.

In addition the this, the libraries that were used can be configured to different degrees - these are further variables that were discarded / left at their default setting due to time constraints.

It would have been interesting to visualize and compare the internal structures of the indices, however this was also not possible due to time constraints.

As everything in the benchmarks was implemented in Rust, which is quite different from the current *default languages*[1] for the types of backend services where this type of mechanism would usually be implemented in, like *Kotlin* or *GO*. These, usually garbage-collected languages behave quite a bit different than *Rust* does. In the beginning this work included comparisons with Java-equivalents of the presented libraries, but since they were difficult to accurately measure and their performance was so abysmal in comparison anyway - they

[1] Consider this an anecdote and not a fact, determining a global / general default is very difficult and opinion and environment based.

were cut. The author heavily suggests using a language that can provide performance in a high throughput environment if performance matters (and if one is already looking into what datastructure is most efficient / fast - it probably does).

Initially, a mechanism to visualise the regions of the prepared queries was started to be implemented, but this was also cut due to time constraints.

Only an excerpt of the 5.4GiB of data was presented in this work as going through everything in detail is not feasible within the given time frame.

The query responses are only primitively validated. Both types of queries only check that the number of returned elements is correct. Theoretically, it is possible that the two indices do not even return the same results. At this time the author relies on the implementation / testing of the library maintainers.

6 Conclusion and Outlook

6.1 Summary

In conclusion, the HPRTree always builds and deletes faster for the tested datasets in addition to being smaller / more space efficient, but it lacks in query performance in all but very few of the tested datasets (and for the queries that remove all items, but there is little point in using a spatial index for such a task). The former was expected, the latter was not, which may warrant future investigation.

6.2 Next Steps

Executing the benchmarks takes a long time, this should probably be remedied.

The behaviour / measurments of the query benchmarks was unexpected and should be investigated further - the expectation was that the cache locality that the HPRTree provides would make querying a lot faster than the pointer chasing necessary for the *R*Tree*.

Another deeper look into the generated data generally seems sensible as only a portion could be taken into account for this work.

There are several parts of the code that are not well aligned where the one side does more work than necessary and the other has to do extra work to undo it again.

To build on this work, it would make sense to extend the benchmarks to other libraries and to include more variables (like the parameters of the indices). Some of the datasets presented are a bit redundant as they display the same performance characteristics as others - removing those would result in lower time requirements to carry out the benchmarks and to do the analysis of the resulting data.

Bibliography

[1] I. Kamel and C. Faloutsos, "Hilbert r-tree: An improved r-tree using fractals," *Proc. Twentieth Int. Conf. Very Large Databases*, Oct. 1999.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, ACM, May 1990. DOI: 10.1145/93597.98741. [Online]. Available: https://doi.org/10.1145/93597.98741.

[3] S. Leutenegger, M. Lopez, and J. Edgington, "Str: A simple and efficient algorithm for r-tree packing," in *Proceedings 13th International Conference on Data Engineering*, 1997, pp. 497–506. DOI: 10.1109/ICDE.1997.582015.

[4] R. K. V. Kothuri, S. Ravada, and D. Abugov, "Quadtree and r-tree indexes in oracle spatial: A comparison using gis data," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '02, Madison, Wisconsin: Association for Computing Machinery, 2002, pp. 546–557, ISBN: 1581134975. DOI: 10.1145/564691.564755. [Online]. Available: https://doi.org/10.1145/564691.564755.

[5] P. PSC and OSGeo. "Postgis - spatial indexing." en. (2023), [Online]. Available: http://postgis.net/workshops/postgis-intro/indexing.html.

[6] M. Aps. "An introduction to spatial indexing." en. (2023), [Online]. Available: https://mapscaping.com/an-introduction-to-spatial-indexing/.

[7] V. Agafonkin. "Leaflet." en. (2023), [Online]. Available: https://leafletjs.com/.

[8] I. E. T. Force. "Ietf rfc 7946 - the geojson format." en. (2016), [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7946.

[9] D. Crockford and C. Morningstar, *Standard ecma-404 the json data interchange syntax*, Dec. 2017. DOI: 10.13140/RG.2.2.28181.14560.

[10] T. R. Team. "Rust programming language." en. (2023), [Online]. Available: https://www.rust-lang.org/.

[11] P. S. Foundation. "Python programming language." en. (2023), [Online]. Available: https://www.python.org/.

[12] T. M. development team. "Matplotlib - visualisation with python." en. (2023), [Online]. Available: https://matplotlib.org/.

[13] datahub.io, *Country Polygons as GeoJSON - Dataset*. [Online]. Available: https://datahub.io/core/geo-countries (visited on 08/17/2023).

[14] Pareto Software, *World Cities Database | Simplemaps*. [Online]. Available: https://simplemaps.com/data/world-cities (visited on 08/17/2023).

[15] Opendatasoft, *Geonames - All Cities with a population > 1000*. [Online]. Available: https://public.opendatasoft.com/explore/dataset/geonames-all-cities-with-a-population-1000 (visited on 08/17/2023).

[16] Matthew Proctor, *Worldwide City Data - Matthew Proctor*. [Online]. Available: https://www.matthewproctor.com/worldwide_cities (visited on 08/17/2023).