Christian-Albrechts-Universität zu Kiel
Institute of Computer Science

Bachelor Thesis

Clustering Methods for Optimizing Index Partitioning

Maximilian Verwiebe

Supervisor:	Prof. Dr. Peer Kröger
Secondary Supervisor:	Dr. Daniyal Kazempour
Department:	Datenbanksysteme und Data Mining
Address:	Olshausenstr. 40, 24098 Kiel, Germany
Submission Date:	September 26, 2025

Institute of Computer Science — Christian-Albrechts-Universität zu Kiel

Abstract

Spatial indexes like R-Trees degrade in performance when the data partitions are unbalanced and their bounding boxes overlap significantly. This thesis presents a capacity-aware clustering pipeline for index partitioning, which starts with a standard clustering algorithm like k-Means and then optimizes the clusters with a post-processing algorithm to minimize overlap and balance their sizes. Oversized clusters are recursively split by KD-style median cuts along the longest axis, while undersized clusters are greedily merged into the neighbor that minimally increases the union MBR volume. This enforces hard constraints on the cluster sizes while minimizing overlap. The optimized clusters are then used as partitions for building an R-Tree, which improves query performance significantly.

An extensive experimental evaluation on synthetic datasets in different dimensions and data distributions shows that the optimization achieves a median reduction of $\approx 8.3\%$ and a mean reduction of $\approx 7.7\%$ in node visits, with positive effects in roughly 60–65% of scenarios. The largest gains occur with linear split heuristics.

Statement of Originality

I hereby declare that I have prepared this thesis on my own and without outside help. I have not used any sources or aids other than those indicated.

The submitted written version of the thesis corresponds to the version on the electronic storage medium.

For formulation and translation support, I used AI-based tools in a very limited way. The research content, analysis, and conclusions are my own.

Furthermore, I assure that this thesis has not yet been submitted as a final thesis elsewhere.

Kiel, September 26, 2025

Maximilian Verwiebe

Contents

Statement of Originality **III**

Contents **IV**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement 1
1.3 Goals and Contributions 2
1.4 Structure of the Thesis 2

2 Fundamentals **3**
2.1 Data 3
2.2 Clustering 3
2.2.1 k-Means 5
2.2.2 Bounding Boxes 7
2.3 Spatial Indexing 7
2.3.1 R-Tree 8
2.4 Queries 10
2.4.1 Nearest Neighbor Queries (kNN) 10
2.5 Evaluation of Spatial Indexes 13
2.5.1 Visited Nodes 13

3 Related Work **14**

4 Problem Context **16**

5 Baseline Clustering with k-Means **18**

6 Post-Processing Cluster Optimization Algorithm **20**
6.1 Goal and Setting 20
6.2 Design Principles 20
6.3 Algorithm Overview 21
6.3.1 Phase 1: Split Oversized Clusters 21
6.3.2 Phase 2: Merge Undersized Clusters 22
6.4 Pseudocode 24
6.5 Termination 26
6.6 Complexity and Scalability 26
6.7 Example 27

6.8 Effect on Indexing 28

7 Integration of Optimized Clustering into Index Partitioning 29

7.1 Pipeline Overview 29
7.2 Basic Clustering 30
7.3 Optimizing the Clusters 30
7.4 Mapping Clusters to R-Tree Partitions 30
7.5 Building the R-Tree 30

8 Evaluation 31

8.1 Research Questions and Setup 31
8.1.1 Test Case 31
8.2 Overall Results (RQ1) 35
8.2.1 Results 36
8.3 Factor Analysis (RQ2) 40
8.4 Decision Rules and Feature Importance 45
8.5 Build Time vs. STR Bulk-Loading (RQ3) 50
8.6 Negative Cases and Causes 52
8.7 Summary 53

9 Summary and Outlook 54

9.1 Key Findings 54
9.2 Limitations 54
9.3 Future Work 54

1 Introduction

The first chapter introduces the motivation, problem statement, goals, and contributions of this thesis. It also provides an overview of the structure of the thesis.

1.1 Motivation

With the continuous growth of digital data sets, the efficient organization of data becomes more and more important. Especially for spatial data, the underlying data structure is crucial when it comes to performance. Spatial data refers to multi-dimensional data. For example, data in geographic information systems, navigation applications, or machine learning use cases. Spatial index structures, like R-Trees, are widely used for that, but they reach their limits when dealing with large amounts of data and uneven data distributions. Overlapping and unbalanced partitions mean that queries have to pass through an unnecessary number of nodes in an R-Tree index, which increases both runtime and I/O costs.

A promising way to avoid these problems is a form of preprocessing the data before indexing it. We can reach this through clustering to create partitions that are more compact, balanced, and better separated from each other.

1.2 Problem Statement

The main problem with R-Trees lies in the way data is inserted into the index. The R-Tree heuristic often produces large, uneven data partitions with big spatial overlaps. These overlaps lead to inefficiencies during querying, as multiple branches of the tree must be traversed.

The aim of this thesis is therefore to develop a method that creates more optimized partitionings in R-Trees.

The central question is: How can clustering methods such as k-Means be used and extended to achieve more efficient partitioning for R-Trees?

1.3 Goals and Contributions

The main objective of this thesis is the development of an algorithm to optimize clusters regarding size constraints and overlap of bounding boxes. Whereas the main contributions of this work are:

- The combination of a k-Means approach with a subsequent optimization process.
- The integration of this process into the R-Tree index building process.
- A comprehensive experimental evaluation based on 1080 test cases.

1.4 Structure of the Thesis

The structure of this thesis is as follows: Chapter 2 introduces the basic concepts of this thesis, including data, clustering, bounding boxes, and R-Trees. Chapter 3 provides an overview of related work. Chapter 4 describes the problem context. Chapters 5 and 6 first explain baseline clustering with k-Means and then the developed optimization algorithm to perform post-processing. Chapter 7 shows the integration of the approach into the structure of R-Trees. Chapter 8 presents the experiments and their results. Finally, Chapter 9 summarizes the key findings and provides an outlook on possible extensions.

2 Fundamentals

In this chapter, we introduce the fundamental concepts and definitions to understand this thesis. We look at the data, clustering, bounding boxes, spatial indexing, R-Trees, and queries.

2.1 Data

The term *data* refers to a collection of information that can be processed and analyzed by a computer. Data can be in various forms, such as numbers, text, images, or audio. A *dataset* is a collection of data, often organized in a structured way.

We consider a dataset $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ where each point $x_i = (x_{i1}, \ldots, x_{id})$ represents data with d numerical features.

Note on numerical features. In this thesis, we focus on numerical features, which are continuous values that can be measured and compared. Categorical features, which represent discrete categories or labels, are not considered in this work. In practice, categorical features are often transformed into numerical representations. For example, one-hot encoding or vector embeddings can be used for that.

Distance metric. Unless stated otherwise, we use the Euclidean distance $\|x - y\|_2$. The choice of distance metric can depend on the application (for example, Manhattan distance).

2.2 Clustering

Clustering is a method of unsupervised machine learning that aims to divide a given dataset into groups, known as clusters, which are disjoint. The special feature of clustering is that there are no predefined labels in the dataset to create such clusters. The used algorithm therefore learns completely independently and unsupervised. Overall, the target goal of clustering is to group objects within a cluster so that they are as similar as possible to each other while being as dissimilar as possible to objects in other clusters.

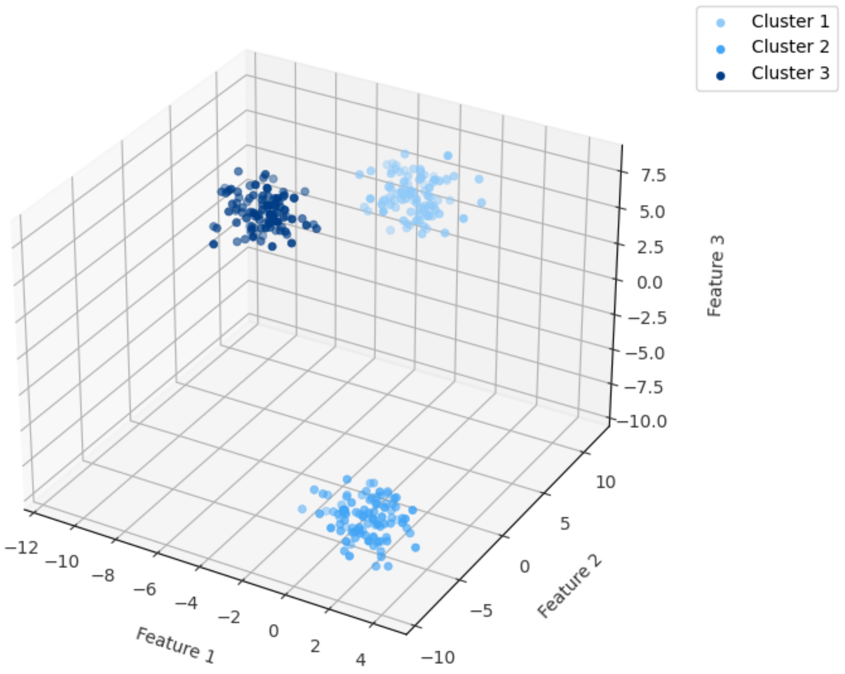


Figure 2.1: Example clustering of data points in 3D space. The algorithm learnt, that the points within each cluster are more similar to each other than to those in other clusters.

In Figure 2.1, the data points have been grouped into clusters, which are represented by the colored circles. The three dimensions represent the three features of the data points, which are used to determine the similarity between them.
A cluster is therefore a set of data points in which the points are similar to each other. The similarity can be defined in different ways, but often it is based on the distance between points.
Clustering methods are used in many areas in practice. It is widespread in marketing for customer segmentation and anomaly detection in financial transactions or network traffic.

2.2.1 k-Means

The k-Means algorithm is one of the most common clustering methods. It was introduced in 1967 by MacQueen. [Mac67] Originally, the term k-Means refers to the underlying optimization problem of partitioning a set of n data points into k clusters in such a way that the points within each cluster are as similar as possible. Formally, this problem is known to be NP-hard, even for $k = 2$ in high-dimensional spaces.

Lloyd's algorithm is a popular method for solving the k-Means problem. It iteratively assigns each datapoint to the nearest cluster centroid and updates the centroids until convergence. [Llo82]

Given:

- A dataset $X = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^d$
- A number of clusters $k \in \mathbb{N}$
- An assignment of points to clusters $C = \{C_1, C_2, \ldots, C_k\}$, where $C_i \subset X$

The objective is to find a partition C that minimizes the following function:

$$L = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2 \tag{2.1}$$

Where:

- $\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$ is the centroid of cluster C_i
- $\|x - \mu_i\|^2$ is the squared Euclidean distance from point x to its centroid

Algorithm 1 Lloyd's Algorithm for k-Means Clustering

```
Require: Dataset $X = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^d$, number of clusters $k$
Ensure: Cluster assignments and centroids $\mu_1, \ldots, \mu_k$
1: Initialize centroids $\mu_1, \ldots, \mu_k$ (e.g., randomly)
2: repeat
3:     for each data point $x_i \in X$ do
4:         Assign $x_i$ to nearest cluster:
               $$c_i \leftarrow \arg\min_j \|x_i - \mu_j\|^2$$
5:     for each cluster $j = 1$ to $k$ do
6:         Update centroid:
               $$\mu_j \leftarrow \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$
7: until convergence
```

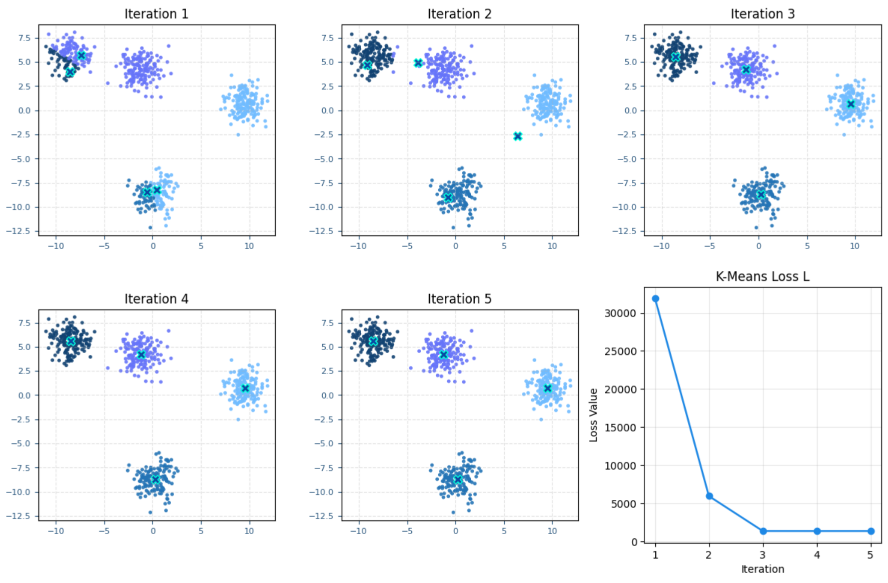


Figure 2.2: Example of k-Means with k=4. Loss function converges at interation 4 and 5, so the algorithm would stop there.

In Figure 2.2, the k-Means algorithm is illustrated with $k = 4$ clusters. The algorithm starts with random centroids (iteration 0) and iteratively refines the clusters and centroids until convergence (iterations 4 and 5).

2.2.2 Bounding Boxes

Given a cluster $S \subset \mathbb{R}^d$, its Minimum Bounding Rectangle (MBR) is

$$\mathrm{MBR}(S) = [\ell_1, u_1] \times \cdots \times [\ell_d, u_d], \quad \ell_k = \min_{x \in S} x_k,\ u_k = \max_{x \in S} x_k.$$

Quality measures. For $d = 2$, the area and perimeter are

$$\mathrm{area}(R) = (u_1 - \ell_1)(u_2 - \ell_2),$$

For two MBRs R and R', the overlap area is

$$\mathrm{overlap}(R, R') = \prod_{k=1}^{2} \max\bigl(0,\ \min(u_k, u'_k) - \max(\ell_k, \ell'_k)\bigr).$$

Figure 2.3 shows an example of bounding boxes in 2D space. The MBRs enclose all points of each cluster.

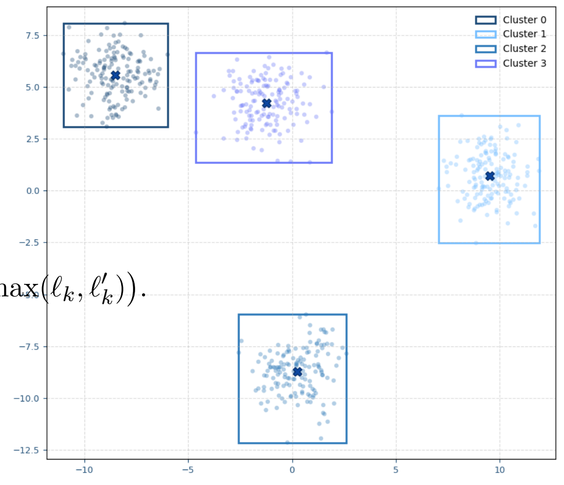


Figure 2.3: Example of bounding boxes in 2D space.

2.3 Spatial Indexing

Indexing refers to the process of organizing and structuring a dataset to make it easier to use it and find data in it. It often involves the usage of complex data structures.

Spatial indexing is used to index multi-dimensional information. These information objects can be rectangles, 3D position vectors, or any arbitrary high-dimensional data.

In such an index, data points are organized in a way that enables efficient spatial queries like:

- Simple point queries: Finding information at a given position
- Range queries: Finding all information within a specified region
- Nearest neighbor queries: Finding the closest information objects to a given location

There exist multiple data structures for spatial indexing, with R-Trees being one of the most popular.

2.3.1 R-Tree

The R-Tree is a tree-based approach for spatial indexing. It is designed for efficient multi-dimensional spatial data. [Gut84] It was introduced in 1984 by Antonin Guttman and quickly became one of the most widely used indexes, especially in databases.
R-Trees are balanced tree structures. Each node represents a minimum bounding rectangle (MBR) of all its child nodes. The leaf nodes contain pointers to the actual spatial data objects.
It uses these two main arguments:

- M = Maximum points per internal node
- m = Minimum points per internal node where $m \leq \frac{M}{2}$

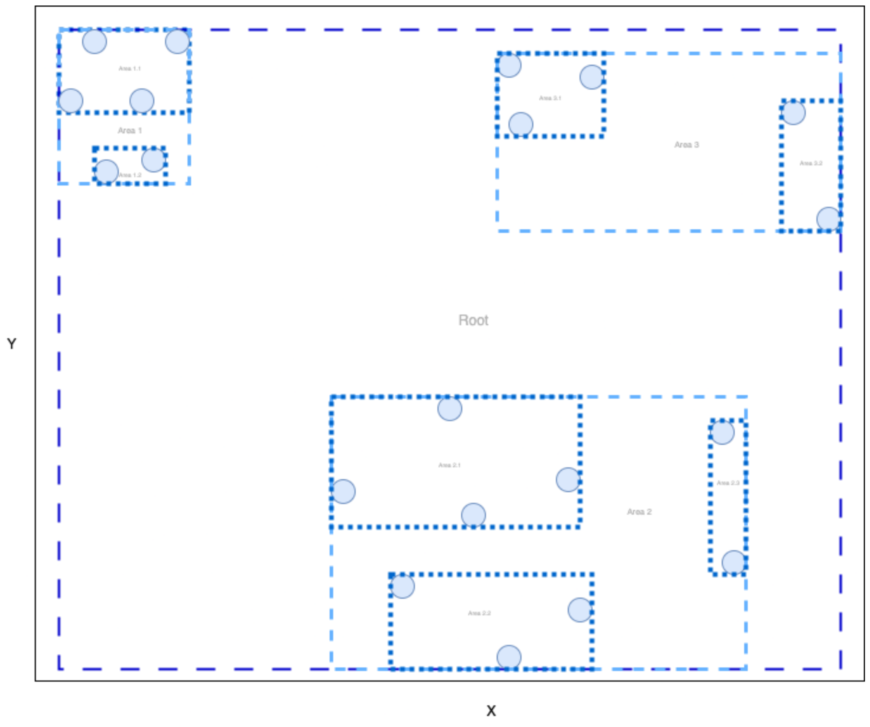


Figure 2.4: Spatial data with R-Tree indexing. ($M = 4$; $m = 2$)

One of the most common applications are range queries and nearest-neighbor queries. The R-Tree allows large parts of the tree, that don't overlap with the query range to be pruned. This significantly improves the query performance compared to linear scans. That's the reason why R-Trees are so popular in data management and sciences.
Typical (internal) operations of an R-Tree are:

- Insertion: The R-Tree inserts an object as leaf node into the MBR, which leads to the least enlargement.

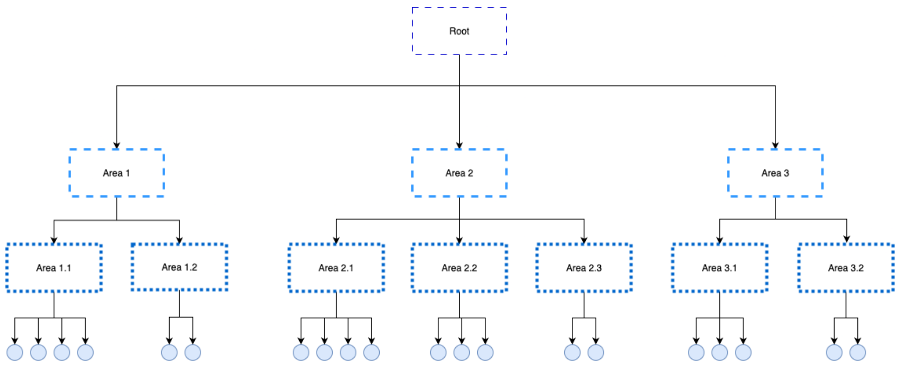


Figure 2.5: R-Tree representation of spatial data. ($M = 4$; $m = 2$)

- Node overflow: If the parameter M is exceeded for a node, it is split into two nodes, using a split heuristic (linear, quadratic...).
- Deletion: On deletion the R-Tree structure might need to be rebalanced, depending on m.

Figures 2.4 and 2.5 show an example of spatial data with R-Tree indexing and its tree representation.
An R-Tree is usually built with the following split heuristics.

Linear Split This split heuristic selects two entries as seeds that are the most distant from each other. The remaining entries are then assigned to the two seeds, based on which seed's MBR enlargement is smaller. This method is simple and fast but can lead to unbalanced trees with large MBRs. The runtime complexity is $O(n)$.

Quadratic Split This method is more complex. The seeds are selected based on the two entries that maximize the area of the MBR. In other words, the two seeds would waste the most space if placed in the same MBR. The remaining entries are then assigned to the two seeds, based on which seed's MBR enlargement is smaller. This method leads to more balanced trees but is more complex and slower than the linear split. The runtime complexity is $O(n^2)$.

R*-Tree The R*-Tree is not just a single heuristic but a whole extension of the R-Tree based on Guttman's original model. It uses a technique called ”reinsertion” / ”repeated insertion” to improve the quality of the tree. On construction, it tries to insert entries multiple times to find the best possible position. This leads to better clustering and therefore to better MBRs with less overlap.

2.4 Queries

Queries in spatial indexes are used to retrieve data based on spatial relationships.

A naive approach would be to calculate the distance from the query point to all points. This would be very expensive for each query, especially for large datasets, because it is in $O(n)$. R-Trees allow pruning of large parts of the search area, which leads to much faster queries.

2.4.1 Nearest Neighbor Queries (kNN)

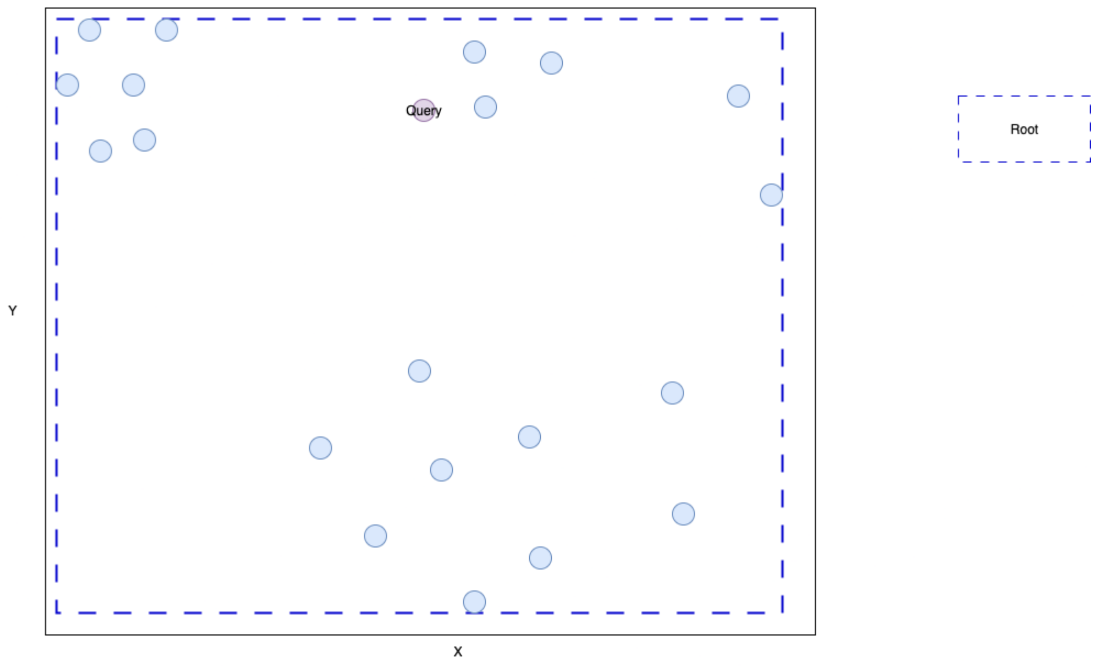
Nearest Neighbor Queries (kNN) are used to find the k closest points to a given query point, where $k \in \mathbb{N}$. The idea of the kNN algorithm was introduced by Cover and Hart in 1967. [CH67] This thesis works with the kNN-Index-APL algorithm, which is a best-first search algorithm for kNN queries on R-Trees. It was found 1995 by Roussopoulos, Kelley and Vincent. [RKV95]
There are many metrics for deciding how close two points are in space. The most common ones are:

- Euclidean distance: The straight-line distance between two points in space.
- Manhattan distance: The sum of the absolute differences of the coordinates.

The choice of metric depends on the application and the data.
It works by using a spatial index to quickly eliminate large areas of the search space that do not contain any potential nearest neighbors.

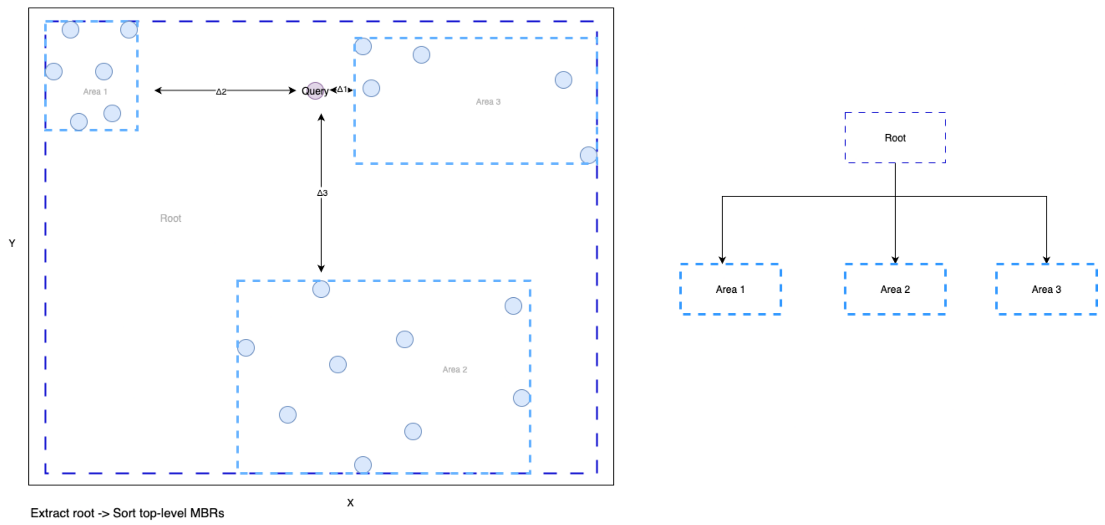
Algorithm 2 kNN-Index-APL(pa, q, k) Best-First Search

```
Require: Spatial index root node $pa$, query point $q$, number of nearest neighbors $k$
Ensure: Result set $R$ containing the $k$ nearest neighbors of $q$
 1: Initialize priority queue $Q$ and insert $pa$ with key MINDIST($pa, q$)
 2: Initialize result set $R \leftarrow \emptyset$
 3: repeat
 4:     Remove entry $e$ from $Q$ with smallest MINDIST($e, q$)
 5:     if $e$ is an internal node then
 6:         for each child entry $f$ of $e$ do
 7:             Insert $f$ into $Q$ with key MINDIST($f, q$)
 8:     else if $e$ is a data object then
 9:         Add $e$ to $R$
10:         if $|R| > k$ then
11:             Remove the farthest object from $R$
12: until $|R| = k$ and $\min_{g \in Q}$ MINDIST($g, q$) $\geq \text{DIST}_{\max}(R, q)$
```



PQ: { (Root, 0) }
Result = {}

Figure 2.6: Example kNN-Index-APL with k=3 (1/5)



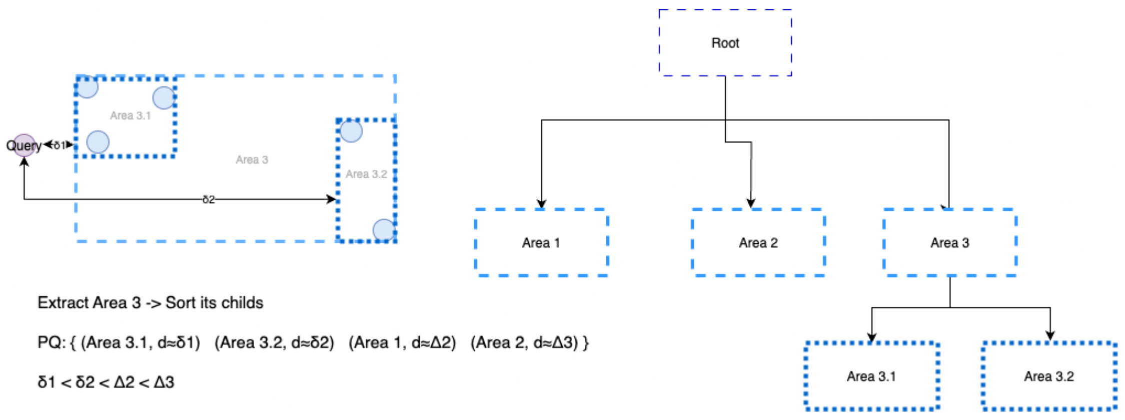
Extract root -> Sort top-level MBRs

PQ: { (Area 3, d≈Δ1) (Area 1, d≈Δ2) (Area 2, d≈Δ3) }

Δ1 < Δ2 < Δ3

=> The query is significantly closer to Area 3 than to the other two areas.

Figure 2.7: Example kNN-Index-APL with k=3 (2/5)

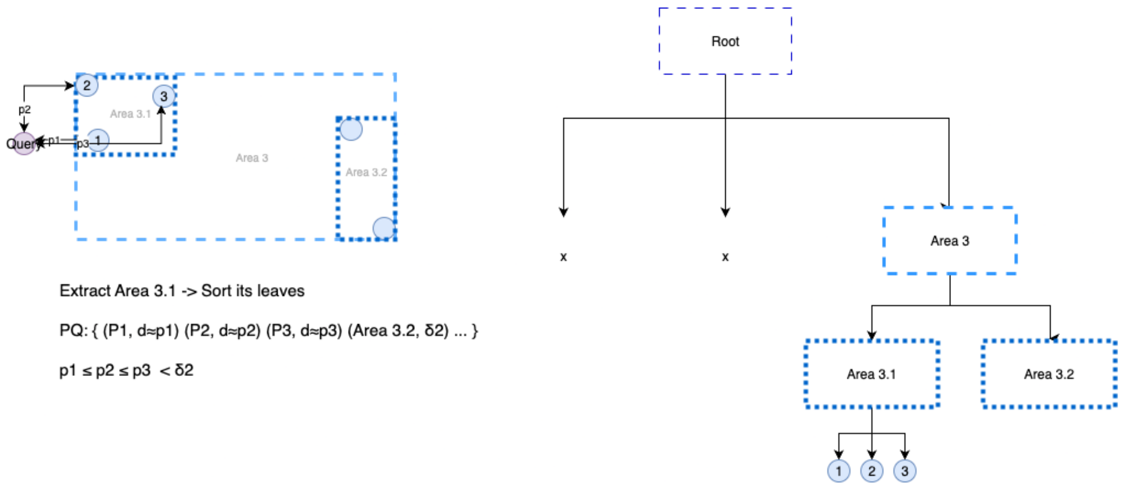


Extract Area 3 -> Sort its childs

PQ: { (Area 3.1, d≈δ1) (Area 3.2, d≈δ2) (Area 1, d≈Δ2) (Area 2, d≈Δ3) }

δ1 < δ2 < Δ2 < Δ3

Figure 2.8: Example kNN-Index-APL with k=3 (3/5)

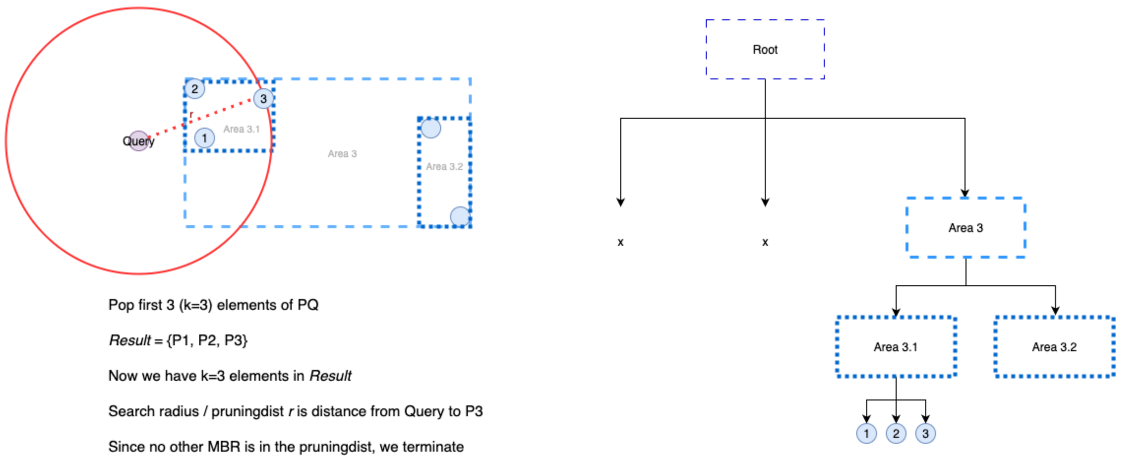


Extract Area 3.1 -> Sort its leaves

PQ: { (P1, d≈p1) (P2, d≈p2) (P3, d≈p3) (Area 3.2, δ2) ... }

p1 ≤ p2 ≤ p3 < δ2

Figure 2.9: Example kNN-Index-APL with k=3 (4/5)



Pop first 3 (k=3) elements of PQ

Result = {P1, P2, P3}

Now we have k=3 elements in *Result*

Search radius / pruningdist *r* is distance from Query to P3

Since no other MBR is in the pruningdist, we terminate

Figure 2.10: Example kNN-Index-APL with k=3 (5/5)

Figures 2.6, 2.7, 2.8, 2.9 and 2.10 show an example of a kNN query with $k = 3$ on an R-Tree. We start with the root node in the priority queue (Figure 2.6). We then iteratively explore the nodes with the smallest minimum distance to the query point. We extract the root node and insert its children into the priority queue (Figure 2.7) sorted by distance to the query point. Next, we extract the node with the smallest distance (Figure 2.8) and insert and sort its children into the priority queue (Figure 2.8). In Figure 2.9, after another extraction, we find leaf nodes (P1, P2, P3), which we add to our priority queue. Finally, in Figure 2.10, we pop the first three elements from the priority queue, which are our $k = 3$ nearest neighbors. We insert them into our result set and return it. The pruning distance is the distance to the farthest point in our result set (P3). Hence, no other node in the priority queue can be closer to the query point than P3 (pruning distance), the algorithm terminates.

2.5 Evaluation of Spatial Indexes

One of the core concepts of a Spatial Index is the fast indexing of data, which means that the index structure should be able to process the above-introduced queries fast. Therefore, our most important metric for evaluating the performance of a spatial index, or more precisely an R-Tree, is how quickly it finds data.

2.5.1 Visited Nodes

We test the R-Tree performance by counting the number of nodes visited during a kNN query.
For a query point $q \in \mathbb{R}^d$ and $k \in \mathbb{N}$ and an R-Tree index I , let

$$V_I(q) \in \mathbb{N}$$

be the *number of visited nodes* during a kNN query on the current index. We define the *number of visited nodes* as the traversal cost needed to reach from the root to the kNN results (leafs) in the tree structure for q.

3 Related Work

Now that all the basic concepts are familiar and the aim of the thesis is clear, let's take a look at the current state of research and related work. This includes various R-Tree variants, bulk loading methods, high-dimensional indexing approaches, machine-learned methods, and constrained k-Means clustering.

R-Tree and R*-Tree. Guttman's R-Tree is the classic dynamic index structure for spatial data. It forms the basis for many subsequent spatial indexing methods, including the R*-Tree. As already explained in section 2.3.1 R-Tree, the R*-Tree improves splitting and uses reinsertion to reduce the area, perimeter, and overlap of the MBRs, therefore optimizing the R-Tree for better query performance. Especially the R*-Tree is still a robust standard in many systems.

Bulk loading and space-filling orders. For static or batched data, bulk loading methods can create R-Trees with better performance than those built by inserting objects one by one. These methods typically involve sorting the data according to a space-filling curve before constructing the tree. The Hilbert R–Tree sorts objects by Hilbert values before inserting them into the R-Tree [KF94]. Another form of bulk loading is the STR (sort-tile-recursive) algorithm. It's a simple method that tries to generate well-filled nodes with little overlap [LLE97].

High-dimensional indexing with X-Trees. As dimensionality grows, MBRs get bigger and overlap increases, which leads to a decrease of pruning capabilities due to the curse of dimensionality. The X–Tree addresses this by using overlap-avoiding splits and, when necessary, using supernodes instead of forcing poor splits, which improves query performance in moderate-to-high dimensions [BKK96].

Machine-learned bulk loading. PLATON uses a top-down R–Tree construction and applies Monte Carlo Tree Search to learn a partitioning policy fine-tuned to the data and workload. This even outperforms classical bulk loaders (including R*-Trees) in query performance while retaining the R-Tree structure [YC23].

Constrained K-Means Clustering. Because page and leaf capacities are limited, clustering must often respect lower/upper bounds on cluster sizes. Constrained k–Means provides a good basis for enforcing *MIN/MAX points per cluster* [BBD00].

Synthesis for this thesis. The literature indicates three effective levers for R–Tree performance:

- *low-overlap partitions* (R*, Hilbert/STR, X–Tree)
- *data/workload-aware construction* (learned and RL-based methods)
- *capacity-aware clustering* that aligns cluster sizes with leaf capacity

This thesis follows these aspects by using k-Means clustering with a capacity-aware post-processing step to create low-overlap partitions, which are then used to build an R-Tree index. This approach uses the strengths of clustering and optimization to improve R-Tree index partitioning and query performance.

4 Problem Context

This chapter describes the problem context of this thesis in more detail.

In many applications, data is stored in large databases, which means that efficient access to this data is crucial. In this context, such spatial data is often stored in spatial structures like R-Trees, which allow very efficient querying and searching. But in order to achieve this efficiency, the data must be partitioned in a way that the resulting clusters are small in size and have a low overlap to each other. R*-Trees are optimized for this purpose, but they are very expensive in dynamic scenarios, where the data is frequently updated. This is where the problem of partitioning arises, which is the focus of this thesis.

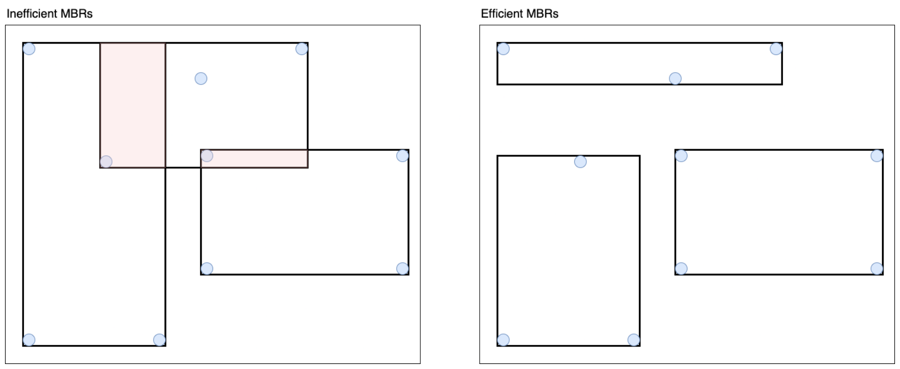


Figure 4.1: Inefficient MBRs (left) show large overlaps. Queries in the overlapping area must search through multiple subtree structures, which increases I/O and CPU time and prevents pruning. Efficient MBRs (right) minimize overlap and area, significantly reducing the number of node visits and false positives.

Figure 4.1 illustrates the problem of inefficient MBRs (Minimum Bounding Rectangles) in an R-Tree. On the left side, we see three large MBRs with significant overlap, which leads to inefficiencies during querying. If a query, for example a kNN query, is executed on such an R-Tree, and the query point is located in the overlapping area (marked in red) of two or more MBRs, the query must search through all MBRs that overlap with the query point. This means that the query must traverse all affected subtrees. This doubles (or multiplies) the number of accesses on the disk. Another problem with very large MBRs is that they cover a large empty space, which results in even more I/Os. Not to be overlooked is the fact that overlap at one level often causes overlap further down. This means the inefficiency multiplies across the whole tree height.

Data pages and capacity. Another problem occurs when we look deeper. In many systems, data is stored on data pages with a fixed size (for example, 4–16 KB). A data page is a physical block of memory or storage. Leaf entries of the R–Tree live on these pages. When we insert points randomly and do not control page capacity (by clustering and optimizing beforehand), several things go wrong. Pages in the leaves tend to be underfilled. When a leaf overflows, it splits, and the two new pages are usually only about 50–70percent full. Over time the average fill level stays well below the page limit, so we need more leaf pages to store the same data. Each leaf split can also push an overflow to its parent, and that can continue up to the root.

This leads to MBRs getting bigger and more overlap. Every split or enlargement forces parent rectangles to grow to cover new children. With random inserts they expand in many directions, depending on the dimension d, so overlap rises and the index must visit more nodes during a simple query. This also destroys the physical locality. Points that are close in the spatial sense are not written close together on a disk page, because they land on different pages created at different times. Even a small kNN query can then touch many pages, which means more random I/O. All of this leads to read/write overloads. More leaves mean more page reads per query, and more splits mean more page re-allocations with underfilled pages.

In short, this all leads to half-empty pages, larger overlapping MBRs, taller trees, and scattered data on different pages, and that directly increases how many nodes and pages each query must visit.

5 Baseline Clustering with k-Means

To solve these problems, as a first step, we want to find k clusters in our dataset, which are going to be improved and optimized by our optimization clustering algorithm in the next section. To create this fixed amount of clusters, we are using the standard k-Means algorithm introduced in Chapter 2.

Selection of Parameters

To manipulate the behavior of the algorithm, we have to chose its parameters.

- **Number of clusters k.** We choose an initial k_0 from storage constraints,

  $$k_0 = \left\lceil \frac{N}{target\_leaf\_size} \right\rceil, \qquad target\_leaf\_size \approx \text{MAX\_POINTS},$$

  so that the generated clusters are approximately going to fit into one leaf (one data page).

- **Initialization.** k-Means
- **Distance / scaling.** Squared Euclidean distance
- **Stopping.** Max iterations: 300
- **Reproducibility.** Fixed random seed per test case

Analysis of Resulting Clusters

k-Means generates some very typical clusters.

Shape The shape of the clusters generated by k-Means is convex. In lower dimensions, the resulting clusters are therefore more spherical or hyperspherical in higher dimensions. This is mainly because k-Means calculates the relationship from a point to a cluster based on Euclidean distance to the centroids, rather than density. This leads the algorithm to not be suitable in every situation. For example, if the points are distributed in ring structures, k-Means fails.

Size and density The clusters generated by k-Means can vary in size a lot. By size we mean the number of points in a cluster, the cardinality. k-Means has no upper or lower limit for cluster sizes. But k-Means has an implicit tendency to form clusters that cover regions of roughly similar spatial size. This is due to the fact that k-Means tries to minimize the variances within a cluster. But this does not mean that clusters contain a similar amount of points. The only thing to control is the number of clusters k to generate.

Example: In a distribution, if one start cluster is large in space but barely populated with points and another is compact and dense, k-Means treats both clusters ”equally” because it only minimizes distances to the center. This will result in clusters that cover similarly sized regions in space but can contain very different numbers of points.

Outliers The k-Means algorithm is very sensitive to outliers. It assigns each point in the dataset to a cluster, even if it might be almost infinitely far away from the cluster centroid.

Impact on the R-Tree Index

What k-Means does is it optimizes the sum of squared distances to centroids. However, an R-Tree benefits from axis-aligned partitions with low overlap. This leads to some mismatches.

Shape mismatch. As mentioned before, k-Means generates convex clusters (spherical), while R-Trees benefit from axis-aligned partitions (rectangles). This can lead to situations where the clusters produced by k-Means do not align well with the boundaries of the R-Tree nodes. The axis-aligned MBR of a spherical cluster is much larger than the true cluster shape. Therefore, we have way more overlap with neighboring MBRs.

Capacity mismatch. k-Means has no lower or upper bound on the number of points per cluster. Some clusters may become very small, while others become very big. We can just try to approximate a perfect cluster count k (see Selection of Parameters).

In short, k-Means gives a fast and stable warm start. Most of the index gains come from the later capacity-aware refinement that tightens box overlap and matches cluster sizes to page capacity.

6 Post-Processing Cluster Optimization Algorithm

Now that we have a basic clustering with k-Means, we want to improve the clusters regarding size constraints and overlap of bounding boxes.

6.1 Goal and Setting

We start with a set of N points $X \in \mathbb{R}^d$ and baseline cluster labels from k-Means. Our goal is to *rebalance* these clusters so that every cluster obeys these capacity constraints

$$\texttt{MIN\_POINTS} \leq |C_i| \leq \texttt{MAX\_POINTS},$$

while also *reducing overlap* between their axis-aligned minimum bounding boxes (MBRs). This aligns clusters with physical data pages (leaves) and improves R-Tree indexing.

Inputs. Points X, initial cluster labels, `MAX_POINTS` (leaf capacity), optional `MIN_POINTS`, random seed.

Outputs. New labels of balanced clusters.

6.2 Design Principles

- **Size constraints.** Force cluster sizes to be within [`MIN_POINTS`, `MAX_POINTS`] to match leaf capacity of the R-Tree index.
- **MBR overlap minimization.** Splits are axis-disjoint along the chosen axis, and merges choose the neighbor cluster with the smallest MBR.
- **Deterministic and fast.** Fully vectorized for performance with fixed seeds for reproducibility.
- **Any dimension.** All steps work in $\mathbb{R}^d$.

6.3 Algorithm Overview

The algorithm has two phases. Phase 1 deals with `MAX_POINTS` violations, while Phase 2 handles `MIN_POINTS` constraints for each cluster.

6.3.1 Phase 1: Split Oversized Clusters

Oversized clusters ($|C| >$ `MAX_POINTS`) are split recursively:

1. Choose the axis with the largest span (max–min).
2. Split at the median along that axis.
3. Check children and split them until each part has $\leq$ `MAX_POINTS` points.

This guarantees the two child clusters are *separated along the split axis*, which immediately reduces overlap of their MBRs. If there is no span, because of, for example, all points being equal, we fall back to a random split to ensure progress of the algorithm.

Note on KD-split. This splitting algorithm is inspired by KD-Trees. KD-Trees recursively partition data along the longest axis, which is efficient for uniform distributions. Here, we use it to ensure that clusters are split into compact, disjoint partitions that fit into a single data page. A similar process was already researched in 2008 by Arge et al in *The Priority R-Tree: A Practically Efficient and Worst-Case-Optimal R-Tree.* [Arg+08]

Now we go into a more detailed explanation.

When do we split? A cluster C is oversized if $|C| >$ `MAX_POINTS`. Such a cluster must be split into smaller parts until every part fits.

How do we choose the split? Let the points of C be $x^{(1)}, \ldots, x^{(m)} \subset \mathbb{R}^d$. For each dimension/axis $k \in 1, \ldots, d$ compute the *span*

$$\text{span}_k(C) \;=\; \max_i x_k^{(i)} \;-\; \min_i x_k^{(i)}.$$

Choose the axis with the largest span:

$$s^\star \;=\; \max\bigl\{\text{span}_k(C) \;:\; k = 1, \ldots, d\bigr\}, \qquad k^\star \;\in\; \bigl\{k : \text{span}_k(C) = s^\star\bigr\}.$$

Now split C at the *median* value $m^\star$ of the $k^\star$-dimension:

$$m^\star \;=\; \text{median}\bigl\{x_{k^\star}^{(i)} \;:\; x^{(i)} \in C\bigr\}.$$

Form two children

$$C_L = \{x \in C : x_{k^\star} \leq m^\star\}, \qquad C_R = \{x \in C : x_{k^\star} > m^\star\}.$$

Result For the two new clusters C_L and C_R we get $MBR(C_L)$ and $MBR(C_R)$ that are *disjoint along the split axis* $k^\star$. This immediately reduces overlap on axis $k^\star$ and can also shrink the total volume of the MBR.

What happens if the cluster is still oversized? If a cluster is still oversized after the split, we push it back into the queue of clusters still to be split. Eventually the algorithm will pop the cluster and perform a split again.

Edgecase zero-span If $\max_i x_k^{(i)} - \min_i x_k^{(i)} = 0$ applies to all dimensions / axes k, we have a special case. It would mean that all points in C are numerically equal. This leads to a median split along any axis k to be wrong, as it would not create any meaningful partitioning. As a fallback, we shuffle the points in C randomly into two partitions of the sizes $\lfloor m/2 \rfloor$ and $\lceil m/2 \rceil$ (with $m = |C|$). So our two cluster partitions look like this:

$$|C_L| = \left\lfloor \frac{m}{2} \right\rfloor, \qquad |C_R| = \left\lceil \frac{m}{2} \right\rceil,$$

A tradeoff is that $MBR(C_L)$ and $MBR(C_R)$ may almost be equal. But the main goal of the algorithm is to satisfy `MAX_POINTS` and `MIN_POINTS`, not to optimize the overlap of every single partitioned cluster. Still, most clusters' MBRs will still be disjoint along the split axis, as this is just a super rare edge case.

6.3.2 Phase 2: Merge Undersized Clusters

If `MIN_POINTS` is set, undersized clusters are greedily merged into a neighbour that minimally increases the combined MBR: After a merge, capacities are checked again. If a merged cluster now exceeds `MAX_POINTS`, it is queued for splitting again. This keeps all clusters within bounds while keeping the bounding boxes tight.

When do we merge? If `MIN_POINTS` is given, a cluster C is *undersized* if $|C| <$ `MIN_POINTS`. Such clusters must be merged into neighbours until all clusters satisfy the lower bound.

Which neighbour do we choose? For a given undersized cluster C_s we pick the target C_t that minimally enlarges the volume of the bounding box of both clusters combined $\text{MBR}(C_s \cup C_t)$. This heuristic is chosen because it aims to keep the resulting new cluster as compact as possible, which serves as a good indicator for minimizing future overlaps with other clusters (MBRs).

$$\text{cost}(C_t) \;=\; \text{vol}\big(\text{MBR}(C_s \cup C_t)\big).$$

Let

$$c_{\min} \;=\; \min_{t \neq s} \text{cost}(C_t).$$

Pick any neighbour $C_t^\star$ with $\text{cost}(C_t^\star) = c_{\min}$, for example.

$$\forall\, u \neq s: \quad \text{cost}(C_t^\star) \leq \text{cost}(C_u).$$

If several neighbours have the same smallest cost, we pick the one with the smallest distance to the centroid of C_s:

$$d_t \;=\; \big\| dist(C_s) - dist(C_t) \big\|_2$$

After we found our target cluster $C_t^\star$, we merge C_s into $C_t^\star$. That means we set $C_t^\star \leftarrow C_t^\star \cup C_s$ and delete label C_s.

What happens if the merged cluster becomes oversized? If the merged cluster $C_t^\star$ exceeds `MAX_POINTS`, we push it back into the queue of clusters still to be split.

What happens if the merged cluster is still undersized? If the merged cluster $C_t^\star$ stays undersized, we keep it in the queue of undersized clusters. At some point it will be popped and merged again with a neighbour, until all clusters satisfy `MIN_POINTS`.

6.4 Pseudocode

Algorithm 3 Post-Processing Cluster Optimization Algorithm

```
Require: $X \in \mathbb{R}^d$ (points), initial labels, MAX_POINTS, optional MIN_POINTS
Ensure: New labels, updated cluster centers/MBRs
 1: Build map clusters: id $\rightarrow$ index-list; compute $R_i$ and $\mu_i$
 2: Queue all clusters with $|C_i| >$ MAX_POINTS
 3: while true do
 4:     while queue not empty do                      ▷ Splitting oversized clusters (KD-split)
 5:         pop cluster $C$
 6:         while $|C| >$ MAX_POINTS do
 7:             get axis with maximum span; split at median into $C_L, C_R$
 8:             replace $C$ by $C_L, C_R$; update $R, \mu$; queue any child still $>$ MAX_POINTS
 9:     if MIN_POINTS is not set then break
10:     if all clusters satisfy $|C_i| \geq$ MIN_POINTS then break
11:     pick smallest cluster $C_s$                    ▷ Merging undersized clusters (greedy)
12:     find neighbour $C_t = arg\min_{j \neq s} \mathrm{vol}(R_s \cup R_j)$     ▷ tie: nearest centroid
13:     merge $C_s \rightarrow C_t$; update $R_t, \mu_t$; delete $C_s$
14:     if $|C_t| >$ MAX_POINTS then queue $C_t$
15: Reindex cluster ids contiguously and return labels and centers.
```

Figure 6.1 shows an example run of the algorithm on a synthetic dataset with `MAX_POINTS` = 75 and `MIN_POINTS` = 25. The process starts with an initial set of four clusters. One cluster is oversized because it contains too many points (n=100). Another cluster is undersized because it has too few points (n=15).
First, the algorithm addresses the oversized cluster. It is split into two smaller parts using the procedure described above. The split is made along the cluster's longest dimension to minimize overlap.
The third phase of the representation shows the result after the split operation. The single large cluster has been replaced by two new, smaller clusters. Now, no clusters are oversized.
Next, the algorithm handles the undersized cluster. The small cluster (n=15) is identified and merged into its best neighboring cluster. The arrow shows which cluster will absorb the small one.
In the final phase, the result after the merge operation is shown. All clusters now meet the size requirements (between 25 and 75 points). The output is a set of balanced clusters with minimized bounding box overlap.

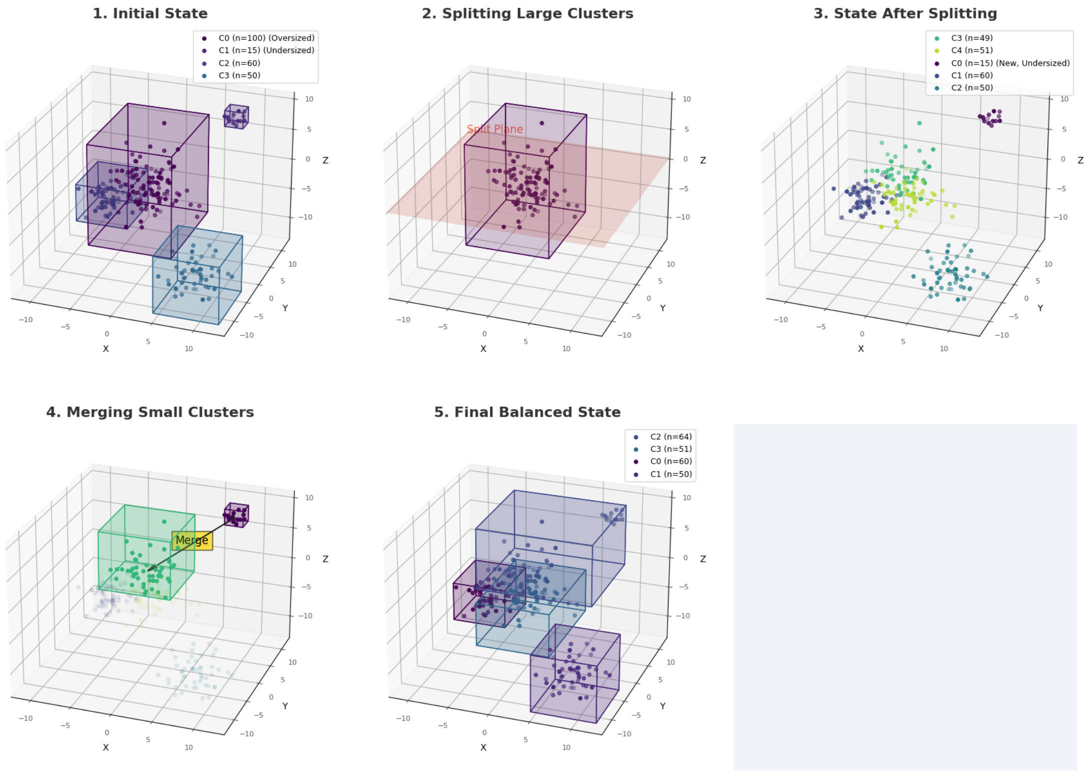


Figure 6.1: Example of the post-processing algorithm with `MAX_POINTS` = 75 and `MIN_POINTS` = 25.

6.5 Termination

The algorithm always terminates. Each split reduces the number of points of a cluster and never increases it. Each merge reduces the number of clusters by 1. There are finitely many points and labels, so the process must stop.

At the end every cluster C_i satisfies the size constraints $\texttt{MIN\_POINTS} \leq |C_i| \leq \texttt{MAX\_POINTS}$.

6.6 Complexity and Scalability

Let N be the number of points, d the dimension, and let $B = \texttt{MAX\_POINTS}$. We assume the input comes with prelabels (initial clusters) from k-Means.

Initialization. Building the index lists and computing one bounding box and one centroid per initial cluster costs

$$O(Nd) \quad \text{time}, \qquad O(N + Td) \quad \text{space},$$

where T is the current number of clusters.

Splitting. For a cluster of size m, each split picks the axis by computing the span for each axis in $O(md)$ and partitions at the median in $O(m)$ (via `np.partition`). The subtree height is $\lceil \log_2(m/B) \rceil$. Therefore:

$$\text{cost per cluster } = O\big(md \, \log(m/B)\big).$$

We summarize this over all clusters (with sizes m_c and $\sum_c m_c = N$):

$$\sum_c O\big(m_c d \, \log(m_c/B)\big) \; \leq \; O\big(Nd \, \log(N/B)\big),$$

with equality in the worst case (all points start in one cluster). Therefore, splitting is *near-linear* in N for constant d and B.

Merging. Let T' be the number of clusters after splitting (typically $T' \approx N/B$). Each merge selects a target by looping over all other clusters and minimizing the union MBR volume. This costs $O(T'd)$ per merge (union of two axis-aligned boxes is $O(d)$). The number of merges is at most T', so the naive bound is

$$O\big({T'}^2 d\big) \quad \text{time}.$$

If a merge creates a cluster that is larger than the capacity B, we put that cluster back into the split queue and split it again with the same KD median rule. Each median split roughly halves the cluster size, so a cluster of size m needs at most $\lceil \log_2(m/B) \rceil$ extra splits to get down to $\leq B$. This means any point is reassigned only a logarithmic number of times. As a result, the extra work caused by such oversize merges stays within the same overall cost as the normal splitting phase, about $O(Nd \log(N/B))$, and the algorithm remains efficient.

Total time. Combining the phases,

$$\boxed{T_{\text{total}} \;=\; O\big(Nd\big) \;+\; O\big(Nd\,\log(N/B)\big) \;+\; O\big(T'^2 d\big)}$$

with $T' \approx N/B$ after splitting:

- If MIN_POINTS is not set $\Rightarrow$ no merges $\Rightarrow$

$$T_{\text{total}} \;=\; O\big(Nd\,\log(N/B)\big).$$

Space complexity. Storing index lists for all clusters and their geometry (MBRs, centroids) needs

$$O(N)\ \text{(indices)} \;+\; O(T'd)\ \text{(MBRs/centroids)} \;=\; O\big(N + (N/B)\,d\big).$$

6.7 Example

Tiny 1D example. Points: $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$, with `MAX_POINTS` $= 4$ and `MIN_POINTS` $=$ 3.

Phase 1 (Split). Split at median $m^\star = 4$:

$$C_L = \{0, 1, 2, 3, 4\}, \quad C_R = \{5, 6, 7, 8, 9\}.$$

Both have 5 points $\Rightarrow$ split again (medians: 2 and 7):

$$\{0, 1, 2\}, \quad \{3, 4\}, \quad \{5, 6, 7\}, \quad \{8, 9\}.$$

Now two clusters are undersized: $\{3, 4\}$ and $\{8, 9\}$ (each $2 < 3$).

Phase 2 (Merge).

$$\text{Step 1: } \{3, 4\} \text{ merge with } \{5, 6, 7\} \;\Rightarrow\; \{3, 4, 5, 6, 7\} \text{ (size 5).}$$

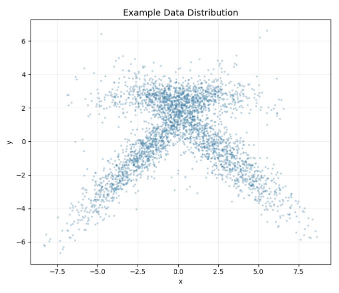


Figure 6.2: Data Distribution

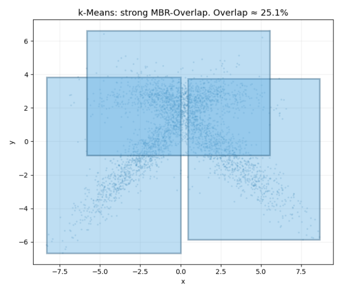


Figure 6.3: k-Means

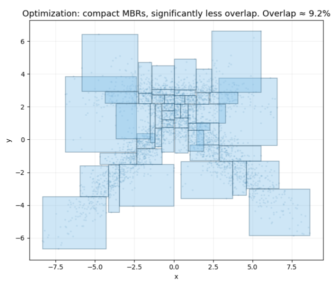


Figure 6.4: Post-Processing

Figure 6.5: Visual impact of the post-processing (optimization). With k-means, few spherical clusters inflate the axis-aligned MBRs and cause strong overlap. The optimization produces many tight boxes and clearly reduces overlap (blue MBRs).

This is oversized ($5 > 4$) $\Rightarrow$ split again at median 5:

$$\{3, 4, 5\}, \quad \{6, 7\}.$$

$$\text{Step 2: } \{6, 7\} \text{ merge with } \{8, 9\} \;\Rightarrow\; \{6, 7, 8, 9\} \text{ (size 4)}.$$

Final clusters:

$$\{0, 1, 2\} \text{ (3)}, \quad \{3, 4, 5\} \text{ (3)}, \quad \{6, 7, 8, 9\} \text{ (4)}.$$

6.8 Effect on Indexing

After this post-processing, cluster MBRs are smaller and overlap less, and cluster sizes fit page capacity. When these MBRs are used as R-Tree partitions, range and kNN queries prune more subtrees and touch fewer nodes, which shows up as fewer visited nodes and lower I/O in our experiments.

Figure 6.5 illustrates the visual impact of the post-processing (optimization) on a dataset. The first image shows the data distribution, the second image shows the MBRs of the clusters produced by k-Means, and the third image shows the MBRs of the clusters after applying our optimization algorithm.

7 Integration of Optimized Clustering into Index Partitioning

Now all the fundamentals and concepts are in place to integrate k-Means clustering and the optimized clustering into the index partitioning process. The goal is to use the clusters generated by the clustering algorithm as partitions for the R-Tree. This means that each cluster will be mapped to a partition/MBR of the R-Tree, and the data points in each cluster will be stored in the corresponding partition.

7.1 Pipeline Overview

Given is a set of data points, which are to be clustered and then used to build an R-Tree. The pipeline consists of the following steps:

- Basic clustering: Apply the clustering algorithm to group similar data points.
- Optimized clustering (post-processing): Usage of the optimized clustering algorithm to refine the clusters, ensuring a minimal overlap of their bounding boxes.
- Partitioning: Map each cluster to a partition/MBR of the R-Tree.
- Index Construction: Build the R-Tree using the clustered data.

Figure 7.1 illustrates the overall pipeline for building an R-Tree with optimized clustering.

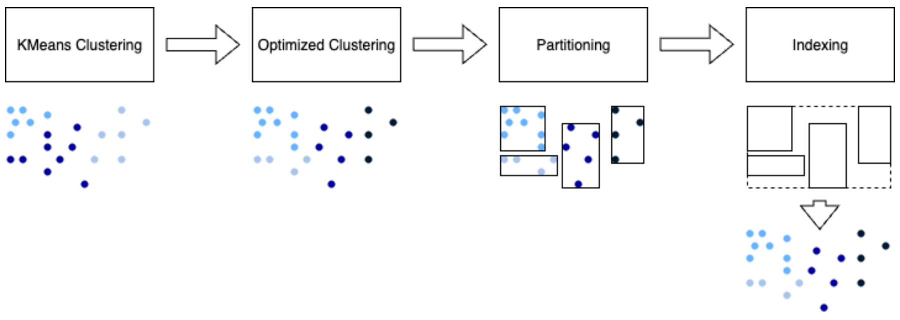


Figure 7.1: Pipeline for building an R-Tree with optimized clustering.

7.2 Basic Clustering

The first step is to apply the k-Means clustering algorithm to the data points. This will group the data points into k clusters based on their spatial similarity.

7.3 Optimizing the Clusters

The next step is to optimize the k baseline clusters generated by the k-Means algorithm. The goal is to have a minimal overlap of the bounding boxes of the clusters and constraints regarding their MIN_POINTS and MAX_POINTS. This is achieved by applying the optimized clustering algorithm defined in Section 5. It finds the best possible partitioning of the data points into clusters, ensuring that the bounding boxes of the clusters do not overlap and that the clusters adhere to the specified size constraints. This process takes the k baseline clusters and generates n optimized clusters.

7.4 Mapping Clusters to R-Tree Partitions

Once the clusters are optimized, the next step is to calculate their MBRs. For each optimized cluster C_i, we compute its axis-aligned MBR.

7.5 Building the R-Tree

Finally, we build the R-Tree by incrementally inserting all points in cluster order. The tree is created with leaf capacity equal to MAX_POINTS, so that each cluster maps to a single leaf whenever $|C_i| \leq \mathit{MAX\_POINTS}$ (otherwise to a small bounded number of leaves). The parent entry over each leaf adopts the leaf's coverage, which closely matches the cluster MBR. This preserves the intended partitioning, we have created earlier. Also, this prevents strong MBR overlaps and improves pruning for kNN queries.

Practical note on bulk-loading. Standard STR bulk-loading as implemented in common libraries, like the C++ library used for this work, libspatialindex, may internally reorder entries and therefore does not guarantee preservation of externally provided cluster partitions. That's why this thesis sticks with the incremental insertion of the MBRs and data points.

8 Evaluation

In this chapter, we present the evaluation of the proposed clustering optimization method for R-Tree index partitioning. We use the pipeline described in Section 7 to build both a baseline R-Tree index and an optimized R-Tree index using the proposed method for a variety of datasets and configurations.

8.1 Research Questions and Setup

To test the effectiveness of the postprocessing clustering algorithm, we use the pipeline described in Section 7. The goal is to evaluate the impact of the optimized clustering on the R-Tree index partitioning.
Mainly, this thesis addresses the following research questions:

RQ1 Does I^{opt} reduce node visits compared to I^{base}? (Query efficiency)

RQ2 Which parameters increase or decrease the effect? (When is it beneficial?)

RQ3 How does I^{opt} compare to STR bulk loading in index construction? (Build time)

... with I^{base} being the baseline index and an optimized index I^{opt}.

We will start with RQ1 and RQ2. We will look at the query efficiency of the optimized index against the baseline index and the most important features.

A testcase is defined as a combination of parameters defined in Table 8.1 that will be used to generate the data points and the R-Tree index.
Since noise applies only to the Gaussian setting, the total number of configurations and therefore test cases is $3 \times 6 \times (1 \times 4 + 1 \times 1) \times 4 \times 3 \times 1 = 1080$.

8.1.1 Test Case

Let a testcase be fixed, with dataset $X = \{x_1, \ldots, x_N\} \subset \mathbb{R}^d$ and two indices built on the same X: a baseline index I^{base} and an optimized index I^{opt} (capacity-aware clustering).

Table 8.2 shows an example configuration for a test case.

Factor	**Levels**
Number of distributions k	{1, 2, 5}
Number of points N	{5,000, 10,000, 20,000, 50,000, 100,000, 200,000}
Data distribution	{GAUSSIAN, UNIFORM}
Noise scale	{0.1, 0.2, 0.4, 0.7} (Gaussian only)
Dimensions d	{2, 5, 10, 20}
R-Tree variant	{0 (linear), 1 (quadratic), 2 (R*)}
Min–max points per distribution	{(0.005, 0.01)} of N
Total combinations	**1080**

Table 8.1: Full factorial design with conditional noise (only for Gaussian).

Parameter	**Value**
Number of distributions k	1
Number of Points N	5,000
Distribution	GAUSSIAN
Noise Scale	0.1
Dimensions d	2
R-Tree Variant	Linear (0)
Min–Max Points	(0.005, 0.01) of N = 25–50 points

Table 8.2: Example test case configuration.

Dataset Generation

The very first step is to generate the dataset for each test case. The datasets' properties depend on the parameters defined in Table **??**. We generate the data for different dimensions d. The data points are generated in two different ways:

- **Gaussian distribution:** The data points are generated from a Gaussian distribution with a given number of clusters k. Each cluster is centered at a random point in the d-dimensional space, and the points are generated around this center with a given noise scale. The noise scale is a parameter that controls the spread of the points around the center.
- **Uniform distribution:** The data points are generated uniformly in the d-dimensional space. The points are generated in a hypercube.

Figure 8.1 shows an example of a dataset with 2 distributions in dimension 2 with Gaussian and uniform distributions.

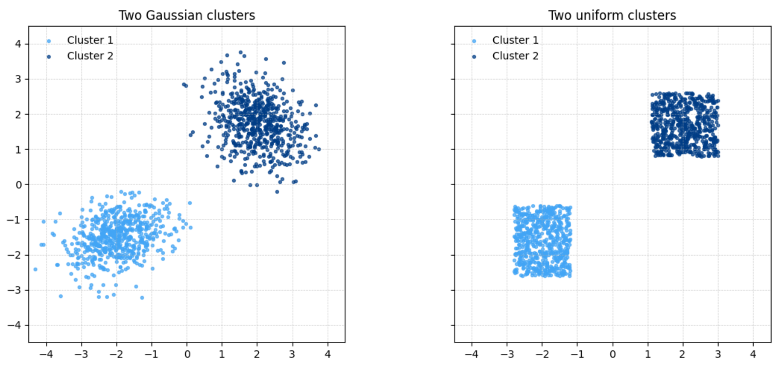


Figure 8.1: Datasets with 2 distributions in dimension 2 with Gaussian (left) and uniform (right) distributions.

Index Construction

We build two R-Tree indices with the same MIN_POINTS and MAX_POINTS for each test case:

- **Baseline index** I^{base}: This index is built by inserting the points of X into an R-Tree without any modifications.
- **Optimized index** I^{opt}: This index is built using the new cluster assignments from the postprocessing algorithm as described in Section 7.5.

Queries and Evaluation Metrics

We test each R-Tree performance by counting the number of nodes visited during a kNN query. For each test case we draw $m = 1000$ i.i.d. queries $q_1, \dots, q_m \sim \mathcal{P}$, where $\mathcal{P}$ equals the generative distribution of X. We set $k = 3$ and evaluate both indices on the same query set.
So in the end we execute the same query twice, once on the baseline index I^{base} and once on the optimized index I^{opt}. The results are then compared to evaluate the impact of the optimized clustering on the R-Tree index partitioning.

The querying of q_j on both indices reduces variance and isolates the effect of the optimized partitioning.

Table 8.3 shows a sample of the results from our 1000 queries for the example test case.

Query Number j	$V_{I^{\text{base}}}(q_j)$	$V_{I^{\text{opt}}}(q_j)$
1	47	32
2	53	41
3	65	48
4	58	39
5	72	59
6	49	42
7	61	45
8	55	38
9	63	52
10	51	40
$\vdots$	$\vdots$	$\vdots$
1000	59	43

Table 8.3: Sample of query results showing node visits for both indices (out of 1000 total queries).

Evaluation

The evaluation is done by comparing the number of visited nodes for each query on both indices. This lets us define simple metrics to compare the two indices:

We analyze the paired differences

$$D(q_j) := V_{I^{\text{opt}}}(q_j) - V_{I^{\text{base}}}(q_j).$$

Table 8.4 shows a sample of the results from our 1000 queries for the example test case, including the difference $D(q_j)$.

Query Number j	$V_{I^{\text{base}}}(q_j)$	$V_{I^{\text{opt}}}(q_j)$	$D(q_j)$
1	47	32	-15
2	53	41	-12
3	65	48	-17
4	58	39	-19
5	72	59	-13
6	49	42	-7
7	61	45	-16
8	55	38	-17
9	63	52	-11
10	51	40	-11
⋮	⋮	⋮	⋮
1000	59	43	-16

Table 8.4: Sample of query results showing node visits for both indices and the difference $D(q_j)$ (out of 1000 total queries).

When $D(q_j) < 0$, it indicates that the optimized index visited fewer nodes for query q_j. Furthermore, we define the following metrics to summarize the performance across all queries for a test case:

- **Mean node visits** $\overline{V}_I = \frac{1}{m}\sum_{j=1}^{m} V_I(q_j)$ measures the average number of nodes visited in index I.
- **Mean difference** $\overline{D} = \frac{1}{m}\sum_{j=1}^{m} D(q_j)$ measures the average absolute improvement in node visits (optimized vs baseline).
- **Median difference** $\widetilde{D} = \text{median}(\{D(q_j)\}_{j=1}^{m})$ is robust against outliers.
- **Relative improvement** $\text{relGain}[\%] = -\frac{\overline{D}}{\overline{V}_{I^{\text{base}}}} \cdot 100$ expresses the percentage reduction in node visits.

A negative $\overline{D}$ and positive relGain indicate that our optimized index requires fewer node visits on average, demonstrating improved efficiency.

8.2 Overall Results (RQ1)

As shown in Table 8.1, we have a total of 1080 test cases. We analyze the results across all test cases to see if there are any patterns or trends.

Table **??** shows a small subset of the 1080 test cases used in our evaluation. Each test case represents a unique combination of the parameters described in Table 8.1.

Test ID	Clusters	Points	Distribution	Noise Scale	Dimensions	R-Tree Variant	Min-Max (Rel.)	Min-Max (Abs.)
case1	1	5,000	gaussian	0.1	2	Linear	(0.005, 0.01)	25.0 - 50.0
case10	1	5,000	gaussian	0.2	2	RStar	(0.005, 0.01)	25.0 - 50.0
case100	1	20,000	gaussian	0.7	10	Linear	(0.005, 0.01)	100.0 - 200.0
case101	1	20,000	gaussian	0.1	10	Quadratic	(0.005, 0.01)	100.0 - 200.0
case102	1	20,000	gaussian	0.2	10	Quadratic	(0.005, 0.01)	100.0 - 200.0
case103	1	20,000	gaussian	0.4	10	Quadratic	(0.005, 0.01)	100.0 - 200.0
case104	1	20,000	gaussian	0.7	10	Quadratic	(0.005, 0.01)	100.0 - 200.0
case105	1	20,000	gaussian	0.1	10	RStar	(0.005, 0.01)	100.0 - 200.0
case106	1	20,000	gaussian	0.2	10	RStar	(0.005, 0.01)	100.0 - 200.0
case107	1	20,000	gaussian	0.4	10	RStar	(0.005, 0.01)	100.0 - 200.0
... (890 more test cases)								

Table 8.5: Sample of test cases from the 1080 total configurations.

Table 8.5 shows only 10 examples from our full test suite of 1080 cases. Each case represents a specific configuration of parameters that could influence R-Tree performance. By systematically evaluating all these configurations, we can identify patterns in the performance improvements offered by our capacity-aware clustering approach.

Test ID	Clusters	Points	Distribution	Noise Scale	Dimensions	R-Tree Variant	Min-Max (Rel.)	Min-Max (Abs.)	$\overline{V}_{I^{\text{base}}}$	$\overline{V}_{I^{\text{opt}}}$	$\overline{D}$	$\widetilde{D}$	relGain[%]	SR[%]
case1	1	5,000	gaussian	0.1	2	Linear	(0.005, 0.01)	25.0 - 50.0	?	?	?	?	?	?
case10	1	5,000	gaussian	0.2	2	RStar	(0.005, 0.01)	25.0 - 50.0	?	?	?	?	?	?
case100	1	20,000	gaussian	0.7	10	Linear	(0.005, 0.01)	100.0 - 200.0	?	?	?	?	?	?
case101	1	20,000	gaussian	0.1	10	Quadratic	(0.005, 0.01)	100.0 - 200.0	?	?	?	?	?	?
case102	1	20,000	gaussian	0.2	10	Quadratic	(0.005, 0.01)	100.0 - 200.0	?	?	?	?	?	?
case103	1	20,000	gaussian	0.4	10	Quadratic	(0.005, 0.01)	100.0 - 200.0	?	?	?	?	?	?
case104	1	20,000	gaussian	0.7	10	Quadratic	(0.005, 0.01)	100.0 - 200.0	?	?	?	?	?	?
case105	1	20,000	gaussian	0.1	10	RStar	(0.005, 0.01)	100.0 - 200.0	?	?	?	?	?	?
case106	1	20,000	gaussian	0.2	10	RStar	(0.005, 0.01)	100.0 - 200.0	?	?	?	?	?	?
case107	1	20,000	gaussian	0.4	10	RStar	(0.005, 0.01)	100.0 - 200.0	?	?	?	?	?	?
... (890 more test cases)														

Table 8.6: Sample of test cases from the 1080 total configurations with performance metrics.

Table 8.6 extends the previous table by including performance metrics for each test case. The metrics include average node visits for both the baseline and optimized indices, mean and median differences in node visits, relative gain percentage, and success rate. These metrics provide a comprehensive view of how the optimized clustering approach impacts R-Tree performance across various configurations.

8.2.1 Results

Now we are looking at the results of the evaluation across all test cases. It's clear to see that in the majority of the test cases, the optimized index has fewer visited nodes for queries than the non-optimized index.

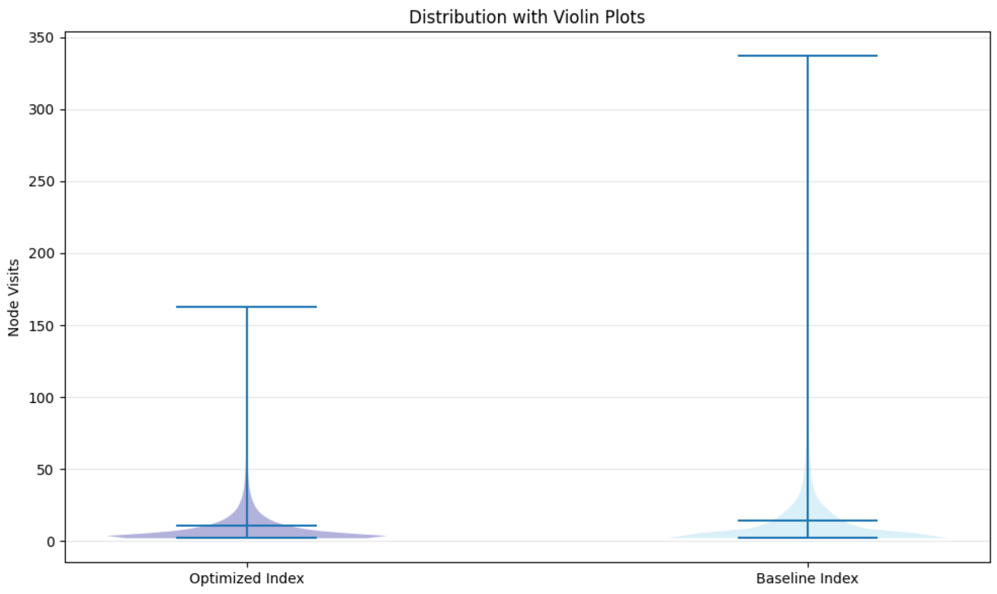


Figure 8.2: Distribution of node visits for baseline index and optimized index. The optimized index shows a lower central trend with a shorter upper tail. In comparison, the baseline index has a significantly higher max number of visited nodes (a longer tail). The interval of node visit counts is therefore larger for the baseline index than for the optimized index.

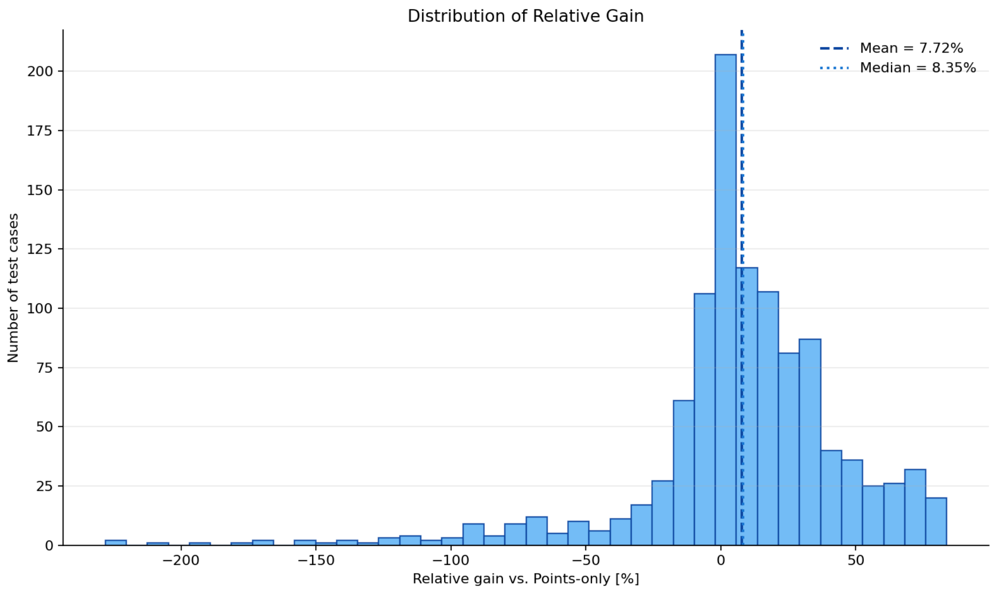


Figure 8.3: Distribution (histogram) of relative gain among all test cases. This histogram shows that the most test cases have a relative improvement in the interval $[-35, 70]$.It also shows the extent of negative outliers. There are a few very rare cases where optimization has led to a negative improvement of up to 200%. However, these outliers are very rare in comparison to all other test cases.

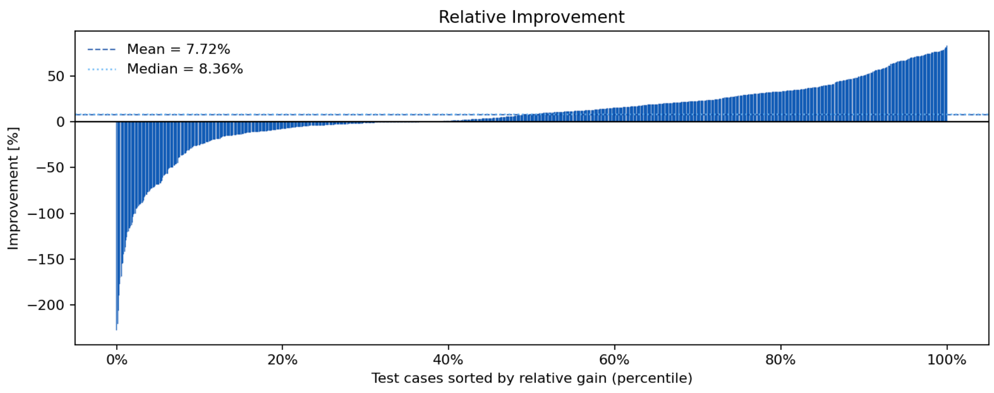


Figure 8.4: This figure shows the relative improvement of the optimization per test case. The test cases are sorted by relative improvement size. In the majority of cases, the optimization has a positive impact. The median of relative improvement is $\approx 8.3\%$, while the mean is $\approx 7.7\%$. This chart also shows a very high variance.

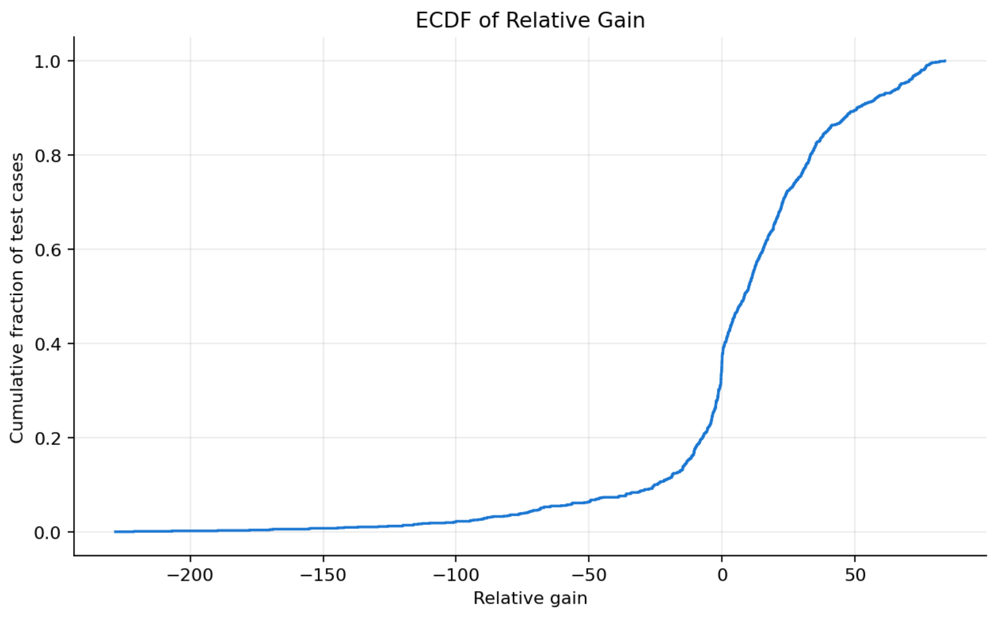


Figure 8.5: Empirical distribution function (ECDF) of relative gains across all test cases. About 60–65% of cases show a positive gain, with a median improvement of ≈8%. The prominent left tail indicates a few large negative outliers. This shows the variability and risk of the optimization on some test cases despite the overall positive effect.

Figure 8.4 (sorted relative improvement) and Figure 8.5 (ECDF) shows that using bounding boxes yields a median relative gain of $\approx 8.3\%$ and a mean gain of $\approx 7.7\%$ across all test cases. The ECDF at $= 0$ indicates that in 60–65% of cases the optimized index outperforms the baseline.

The histogram in Figure 8.3 is asymmetrically distributed. Many cases show small to moderate gains (see the steep ECDF between 0–30%). A long left tail shows a few large drops (down to −230%). Because of these rare outliers with a relative gain of −230%, the mean is below the median. On the positive side, some cases reach gains up to **+80%**.

In Figure 8.2, we see that the optimized index has a lower median and a shorter upper tail. The baseline index has a significantly higher maximum number of visited nodes (a longer tail). The interval of node visit counts is therefore larger for the baseline index than for the optimized index.

To give a brief interim conclusion, optimizing a spatial index with our clustering optimization algorithm is beneficial in the typical case but not universally beneficial. The overall effect is positive, but for a small number of cases, the performance can degrade significantly.

8.3 Factor Analysis (RQ2)

As the next step, we analyze the influence of different parameters on the performance of the optimized clustering algorithm. We are looking at the different features defined in Table 8.1.

Dimensionality

d	Cases	Win rate [%]	Mean rel. gain [%]	Mean diff
2	270	73.0	27.869	-4.171
5	270	47.0	-3.263	-3.324
10	270	55.9	-2.168	-2.919
20	270	74.4	8.460	-4.015

Table 8.7: Performance by dimensionality: number of cases, win rate (share of test cases with fewer node visits), mean relative gain, and mean absolute difference. Mean diff is the average of $\bar{D}$ across all test cases in that dimension group.

Across dimensions, the effect is not uniform (Table 8.7 and Figure 8.6). In $d = 2$, the method wins in most cases (win rate $\approx 73\%$) with the largest average relative gain ($\approx 27.9\%$). In $d = 20$, the win rate is similarly high ($\approx 74\%$) but the average gain is smaller ($\approx 8.5\%$). For $d = 5$ and $d = 10$, the picture is mixed: win rates around 47–56%, and small average effects that can turn negative on average. Overall, lower and very high dimensions benefit more often, while mid-range dimensions show many small wins and a few larger losses.

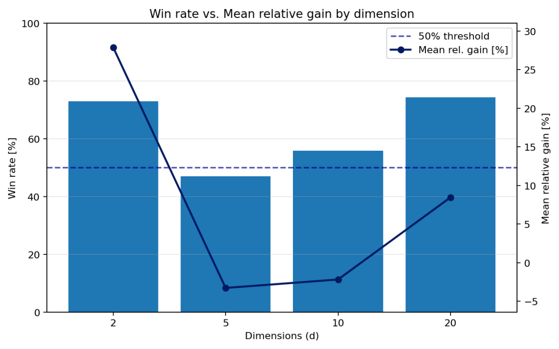


Figure 8.6: Win rates across different dimensions.

We now look at the cases where the optimization has the most negative impact regarding the dimensionality. For this we filter the results for negative relative gains.

The optimization algorithm seems to generate unfavorable effects primarily in medium dimensions (d=5/10), where the

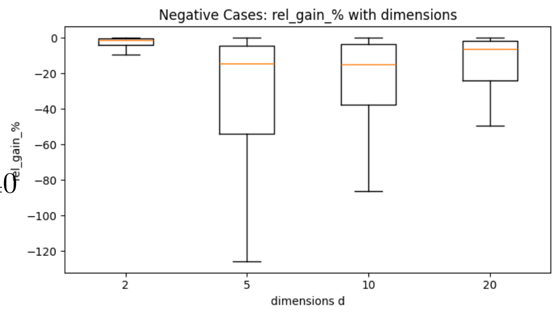


Figure 8.7: Distribution of negative relative improvement (rel_gain_%) according to dimensionality d.

largest losses and greatest dispersion occur. In 2D, the effects are small, while in high dimensionality, the medians approach 0 % again (Figure 8.7).

Number of Points / Points per Distribution

This parameter controls the number of data points in the dataset to be indexed.

Total Points	**Cases**	**Win rate [%]**	**Mean rel. gain [%]**	**Mean diff**
5,000	180	68.9	5.336	-1.452
10,000	180	66.7	7.192	-1.976
20,000	180	66.7	11.012	-3.183
50,000	180	61.7	10.352	-4.159
100,000	180	59.4	9.321	-5.118
200,000	180	52.2	3.134	-5.757

Table 8.8: Performance by number of points: number of cases, win rate (share of test cases with fewer node visits), mean relative gain, and mean absolute difference.

The number of points N in the dataset influences the index size and structure. As the number of points increases, the index becomes larger and potentially more complex, which can impact query performance. The results in Table 8.8 show that smaller datasets (5,000 to 20,000 points) benefit more from the optimization, with win rates around 67–69% and average relative gains up to 11%. The more points are added (50,000 to 200,000), the more the performance tends to degrade, with win rates dropping to 52%.

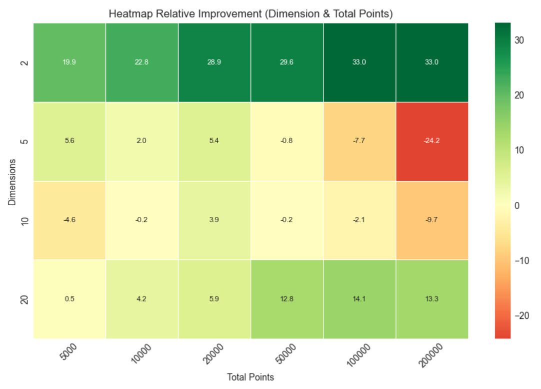
We can also take a look at the number of points per distribution in the dataset. Figure 8.1 shows such a distribution with a number of distributions of 2 and 1000 points per distribution. This is calculated by dividing the total number of points N by the number of distributions k: N/k = Points per distribution.

Points per distribution	Cases	Win rate [%]	Mean rel. gain [%]	Mean diff
1,000	60	76.7	6.074	-1.737
2,000	60	80.0	7.015	-1.730
2,500	60	71.7	6.853	-1.265
4,000	60	80.0	11.548	-2.861
5,000	120	62.5	5.402	-1.738
10,000	180	64.4	9.256	-2.852
20,000	120	60.8	12.252	-4.779
25,000	60	63.3	12.201	-4.539
40,000	60	58.3	8.616	-6.960
50,000	120	52.5	9.246	-4.744
100,000	120	50.8	3.311	-5.106
200,000	60	50.0	-1.454	-4.549

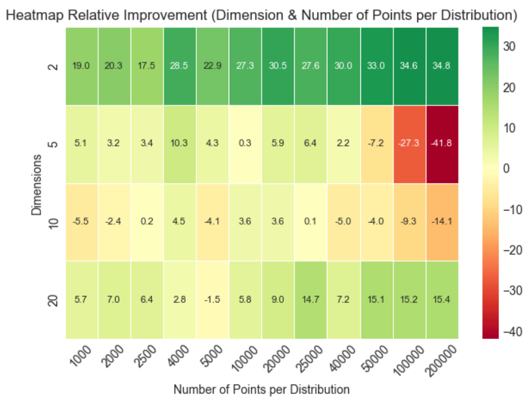
Table 8.9: Performance by points per distribution: number of cases, win rate (share of test cases with fewer node visits), mean relative gain, and mean absolute difference.

In Table 8.9 we can clearly see that the fewer points per distribution, the better the performance in terms of win rate.

Combination with dimensions We have already seen that the number of points and the number of dimensions have a significant impact on the performance of the optimized clustering algorithm. But when we look at the combination of these two parameters in Figure 8.8, we can see that the effect is even more significant.



(a) Relative improvement in relation with the number of total points and dimensions.



(b) Relative improvement in relation with the number of points per distribution and dimensions.

Figure 8.8: Heatmaps showing the relative improvement in relation with the number of points and dimensions. The left heatmap (a) shows the relation with total points, while the right heatmap (b) shows the relation with points per distribution. Both heatmaps indicate that lower dimensions (2D) consistently benefit from optimization, while mid-range dimensions (5D, 10D) show mixed results, especially as the number of points increases. High dimensions (20D) again show more consistent positive gains, particularly with fewer points per distribution.

The heatmaps in Figure 8.8 illustrate the complex interplay between dimensionality and point distribution on optimization performance. In lower dimensions (2D), the optimization consistently achieves positive results across varying point counts, indicating that spatial relationships are more effectively captured in simpler spaces.

Distribution Type and Noise

Distribution	Cases	Win rate [%]	Mean rel. gain [%]	Mean diff
Gaussian	864	66.0	10.215	-3.484
Uniform	216	49.1	-2.237	-4.102

Table 8.10: Performance by distribution type: number of cases, win rate (share of test cases with fewer node visits), mean relative gain, and mean absolute difference.

In Table 8.10 it is clear to see that Gaussian distributions benefit significantly more from the optimization than uniform distributions. Gaussian datasets achieve a win rate of 66% with an average relative gain of 10.2%, while uniform distributions show a mixed performance with only 49% win rate and a slight negative average gain of -2.2%. This suggests that the clustering optimization is more effective when data exhibits natural clustering patterns rather than uniform spread.

Noise scale Noise scale is a parameter that controls the spread of the points around the center in a Gaussian distribution. It is only applied to Gaussian distributions and has no effect on uniform distributions. Therefore a noise scale of 0.0 implies that it is a uniform distribution.

Noise Scale	Cases	Win rate [%]	Mean rel. gain [%]	Mean diff
0.0	216	49.1	-2.237	-4.102
0.1	216	64.4	9.530	-3.099
0.3	216	65.7	10.208	-3.544
0.5	216	68.1	10.026	-3.624
0.7	216	65.7	11.096	-3.668

Table 8.11: Performance by noise scale: number of cases, win rate (share of test cases with fewer node visits), mean relative gain, and mean absolute difference. Note that noise scale 0.0 corresponds to uniform distribution.

The noise scale analysis results in Table 8.11 reveal an interesting pattern. When noise scale is 0.0 (uniform distribution), the optimization performs poorly with only 49.1% win rate and negative average gain, as discussed above. There is no clear difference in performance between the various noise scales (0.1 to 0.7). But the tendency is slightly positive, indicating that some noise may help the optimization algorithm to find better splits and merges.

We now look at the negative cases. The results of this analysis are shown in Figure 8.9. The strongest outliers occur at very low noise levels (0.0 and 0.1). As noise increases, the performance loss does not necessarily become "good", but it does become more predictable. The extreme outliers decrease noticeably.

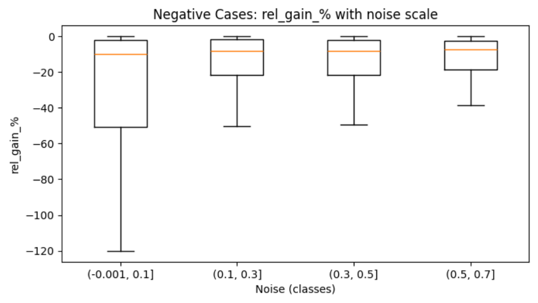


Figure 8.9: Distribution of negative relative improvement (rel_gain_%) according to noise scale.

R-Tree Variant

The last feature to look at, are the different R-Tree variants as defined in Chapter 2.

R-Tree Variant	Cases	Win rate [%]	Mean rel. gain [%]	Mean diff
Linear	360	99.7	38.796	-10.759
Quadratic	360	59.7	-8.596	-0.314
R*	360	28.3	-7.026	0.250

Table 8.12: Performance by R-Tree variant: number of cases, win rate (share of test cases with fewer node visits), mean relative gain, and mean absolute difference.

The analysis proves that the R-Tree variant has a dramatic impact on optimization effectiveness. In Table 8.12 the linear splitting algorithm shows exceptional performance with a 99.7% win rate and nearly 39% average improvement. In contrast, quadratic splitting shows mixed results (59.7% win rate, -8.6% average gain), while R* performs poorly with only 28.3% win rate and negative gains. This suggests that the optimization strategy works best with simpler splitting algorithms that may leave more room for improvement through better clustering.

Figure 8.10 shows that the optimized index performs worst with a quadratic split heuristic in dimension 5. The quadratic variant is also bad in dimensions 10 and 20 but slightly better than in dimension 5.

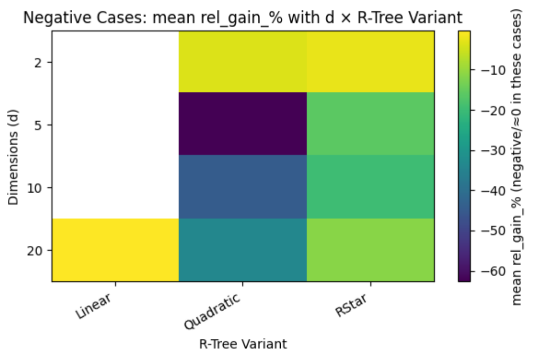


Figure 8.10: Distribution of negative relative improvement.

8.4 Decision Rules and Feature Importance

A feature importance analysis is a statistical technique used to rank features according to their importance in predicting a target variable. In our case, we want to understand which parameters have the most significant impact on the performance of the optimized clustering algorithm.

Decision Tree Classifier

To perform a feature importance analysis, we can use a decision tree machine learning model. Decision trees inherently provide feature importance scores based on how much each feature contributes to reducing the impurity (often the Gini impurity or entropy) in the tree. We can train a decision tree on our dataset and extract the feature importance scores.

At first we have to prepare the data. We define a binary target variable indicating whether the optimization was successful or not:

$$y = bbox\_better = (data['mean\_diff'] < 0)$$

Also, we have to encode the categorical features of our training dataset X (Distribution and R-Tree variant) into numerical values.

Test ID	Dimensions	Points per Distribution	Distributions	Distribution._enc	Noise Scale	Rtree_enc
case1	2	5000	1	0	0.1	0
case10	2	5000	1	0	0.3	2
case100	20	10000	1	0	0.7	0
case1000	20	20000	5	0	0.7	0
case1001	20	20000	5	0	0.1	1

Table 8.13: Sample of training data X with encoded categorical features for decision tree classifier.

A snippet of the training data X is shown in Table 8.13 with Table 8.14 being the encoding scheme for encoded categorical features.

Feature	Encoded Value	Original Value
Distribution	0	Gaussian (864 test cases)
	1	Uniform (216 test cases)
R-Tree Variant	0	Linear (360 test cases)
	1	Quadratic (360 test cases)
	2	R* (360 test cases)

Table 8.14: Encoding scheme for categorical features used in decision tree classifier.

For example, the condition "Distribution (enc) ≤ 0.5" corresponds to Gaussian distribution, while "R-Tree Variant (enc) ≤ 0.5" corresponds to linear splitting. After that, we create a train-test split of the data with 80% training data and 20% test data.

Then we train a decision tree classifier using the parameters $max\_depth = 9$, $min\_samples\_split = 5$ on X_{train} to predict the target label y_{train}. The parameters were selected through trial and error in order to achieve the highest accuracy.
The result is a decision tree with an accuracy of 91.4% on the test set.

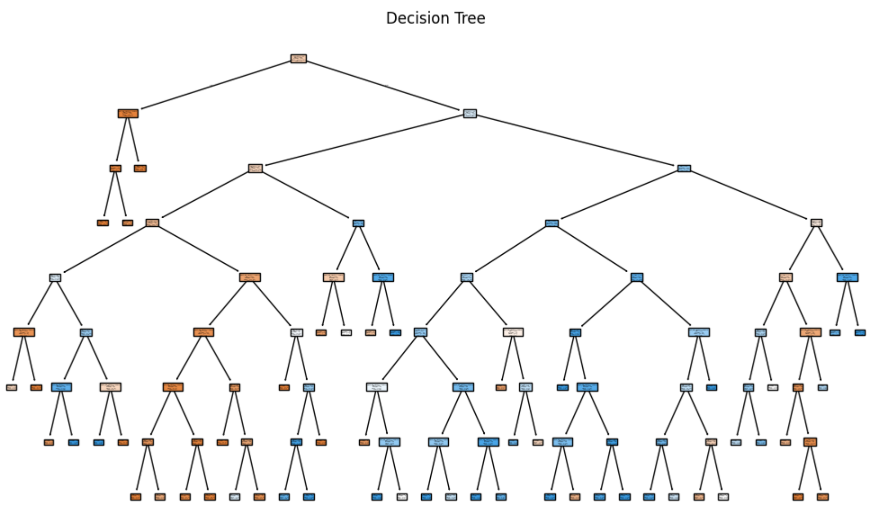


Figure 8.11: Structure of the trained decision tree classifier. Blue means a prediction that the optimized index would perform better, while orange means that the baseline index would perform better.

The structure of the trained decision tree is shown in Figure 8.11. The tree splits the data based on feature thresholds to classify whether the optimized index will perform better than the baseline index. The blue nodes indicate a prediction that the optimized index would perform better, while orange nodes indicate that the baseline index would perform better.

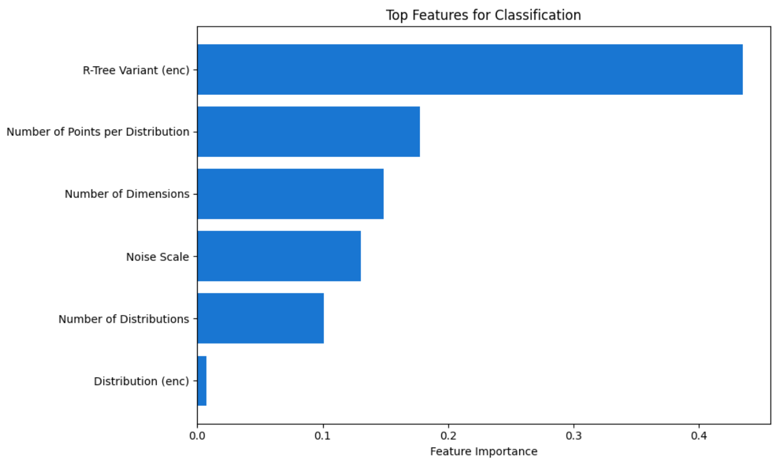


Figure 8.12: Extracted feature importance with scores from decision tree.

The feature importance extraction in Figure 8.12 ranks the features according to variance regarding the relative improvement of using an optimized index. It reveals that the *R-Tree variant* feature is the most important feature. We could have expected that, as the high variance is already visible in **??**. The *Distribution* feature of the dataset (Gaussian or uniform) does not appear to be important at all. It does not seem to have any influence. This also corresponds with the observations from **??**. The features *Number of Points per Distribution*, *Number of Dimensions*, *Noise Scale*, *Number of Distributions* also have an impact, but they are roughly equally important.

We can also derive decision rules from the decision tree that we have trained. In other words, the paths that predict I^{opt}.

rule
rtree_enc <= 0.500 AND points_per_cluster > 2250
pred_class: I^{opt} **confidence:** 1.0 **n_samples:** 229
rtree_enc > 0.500 AND rtree_enc <= 1.500 AND points_per_cluster <= 4500 AND points_per_cluster > 2250
pred_class: I^{opt} **confidence:** 1.0 **n_samples:** 29
rtree_enc <= 0.500 AND points_per_cluster <= 2250 AND dims <= 15
pred_class: I^{opt} **confidence:** 1.0 **n_samples:** 22
rtree_enc > 0.500 AND rtree_enc <= 1.500 AND points_per_cluster > 4500 AND dims <= 3 AND noise > 0.050 AND $n_{\text{distributions}} > 1$
pred_class: I^{opt} **confidence:** 1.0 **n_samples:** 22
rtree_enc > 0.500 AND rtree_enc <= 1.500 AND points_per_cluster <= 4500 AND points_per_cluster <= 2250 AND dims > 7
pred_class: I^{opt} **confidence:** 1.0 **n_samples:** 16
rtree_enc > 0.500 AND rtree_enc > 1.500 AND dims > 15 AND $n_{\text{distributions}} > 3$ AND noise > 0.400

Continued on next page

(Continued)

pred_class: I^{opt} **confidence:** 1.0 **n_samples:** 10

rtree_enc > 0.500 AND rtree_enc <= 1.500 AND points_per_cluster <= 4500 AND points_per_cluster <= 2250 AND dims <= 7 AND dist_enc <= 0.500

pred_class: I^{opt} **confidence:** 0.917 **n_samples:** 12

rtree_enc > 0.500 AND rtree_enc > 1.500 AND dims > 15 AND $n_{\text{distributions}}$ > 3 AND noise <= 0.400 AND points_per_cluster <= 7000

pred_class: I^{opt} **confidence:** 0.875 **n_samples:** 8

rtree_enc > 0.500 AND rtree_enc <= 1.500 AND points_per_cluster > 4500 AND dims > 3 AND dims > 15 AND dist_enc <= 0.500

pred_class: I^{opt} **confidence:** 0.857 **n_samples:** 49

rtree_enc > 0.500 AND rtree_enc <= 1.500 AND points_per_cluster > 4500 AND dims <= 3 AND noise <= 0.050 AND points_per_cluster <= 45000

pred_class: I^{opt} **confidence:** 0.857 **n_samples:** 7

rtree_enc > 0.500 AND rtree_enc <= 1.500 AND points_per_cluster > 4500 AND dims <= 3 AND noise > 0.050 AND $n_{\text{distributions}}$ <= 1

pred_class: I^{opt} **confidence:** 0.765 **n_samples:** 17

rtree_enc > 0.500 AND rtree_enc > 1.500 AND dims <= 15 AND points_per_cluster <= 15000 AND dims <= 3 AND points_per_cluster > 4500

pred_class: I^{opt} **confidence:** 0.75 **n_samples:** 16

The decision tree confirms the feature-importance result. The most important feature is the *R-Tree variant*. Next come *points per distribution* and *dimensionality*. This matches the high variance we already see in 8.12. The feature *distribution* (gaussian vs. uniform) does not that matter much, which also fits 8.10. The features *number of points per distribution*, *number of dimensions*, *noise scale*, and *number of distributions* also have an effect, but they are roughly equally important.
We can also read simple decision rules from the trained tree (paths that predict I^{opt}):

- Linear: if points per distribution > 2250, the tree predicts I^{opt} (confidence 1.0, $n = 229$).
- Quadratic: if $2250 <$ points per distribution ≤ 4500, the tree predicts I^{opt} (confidence 1.0, $n = 29$).
- R*-Tree (niche case): I^{opt} appears mainly when dims > 15, $n_{\text{distributions}} > 3$, and noise > 0.4 (confidence 1.0, $n = 10$).

To summarize this: The index I^{opt} is suitable for a linear R-Tree with large distributions, and for Quadratic R-Trees in the range 2250–4500 points per distribution. Outside these ranges, the benefit is less common and depends more on *number of dimensions* and *noise scale*. The thresholds above are data-driven (picked by the decision tree).

8.5 Build Time vs. STR Bulk-Loading (RQ3)

Additionally to the test cases, we are comparing our approach with the STR-bulk loading method. [LLE97] Bulk loading methods such as STR [LLE97] or Hilbert R-Tree [KF94] show that pre-sorted partitions can significantly increase the efficiency of R-Trees. Therefore, our optimization algorithm and its procedure must be able to keep up in order to be considered successful. For simplicity we compare against the STR algorithm which is a well-known method for bulk loading R-Trees, and its important to understand how our optimization compares to this baseline. We want to look at the building time.

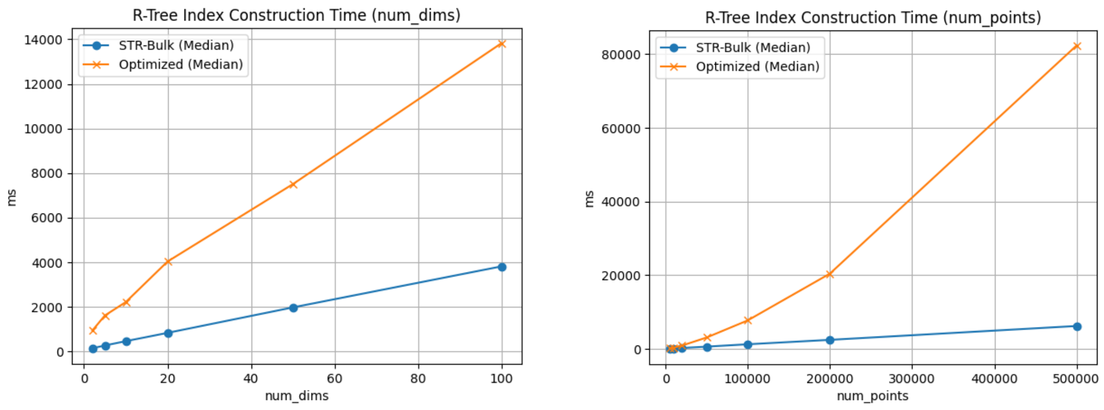
At first we have to get an understanding of the building process of both methods. The optimization of the R-Tree index consists of four main steps:

- Clustering the data points with k-Means in Python3
- Splitting the clusters into smaller groups in Python3
- Computing the bounding boxes in Python3
- Creating the index in C++

Now we generate new test-cases with the following parameters (see Table 8.15).
This time we focus on the impact of the optimization on the building time of the R-Tree index. So we create an optimized index I^{opt} as described above and a baseline index I^{STR} using the STR-bulk loading method for each testcase. We record the building time of both indexes and compare them.

Factor	**Levels**
Number of distributions k	{2}
Number of points N	{5,000, 10,000, 20,000, 50,000, 100,000, 200,000, 500,000}
Data distribution	{GAUSSIAN}
Noise scale	{0.1}
Dimensions d	{2, 5, 10, 20, 50, 100}
R-Tree variant	{0 (linear)}
Min–max points per distribution	{(0.005, 0.01)} of N
Total combinations	**42**

Table 8.15: Full factorial design with conditional noise (only for Gaussian).



(a) Building time comparison in relation with the number of dimensions.

(b) Building time comparison in relation with the number of points.

Figure 8.13: Building time analysis for different methods and dataset sizes.

Figure 8.13 shows that STR–Bulk is significantly faster when building R-Tree indexes, even when dimensionality increases.
Figure 8.14 shows the median build time of the optimized index with its components regarding dimensionality. The total costs are largely determined by k-Means and the R-Tree index construction. As dimensionality increases, the index construction time grows more rapidly almost linear with d, exceeding the k-Means time at around 50–70 dimensions. The optimization of the clusters (Split) and boundng box computation computation steps play a minor role in this. These two feature are almost constant with increasing dimensionality.

Overall, the pure index construction of the optimized approach scales approximately linearly in dimensionality, but with a significantly higher slope than the STR bulk approach.

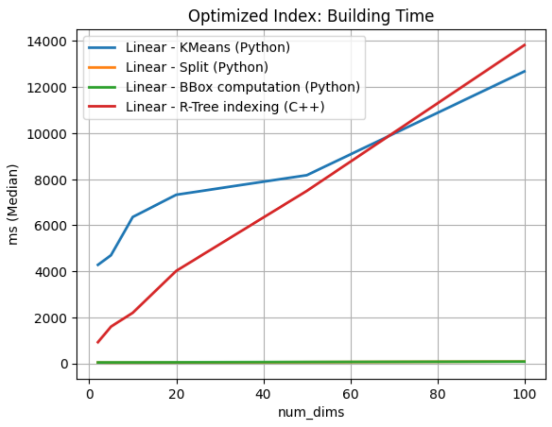


Figure 8.14: Median build time of the optimized index decomposed into components (KMeans, split, BBox calculation, and C++ index construction) as a function of dimensionality.

8.6 Negative Cases and Causes

We have identified cases, in which losses occur often:

- in medium dimensions $d \in \{5, 10\}$ (see Table 8.7)
- with *uniform* or very low-noise distribution (noise $\in \{0.0, 0.1\}$; see Tab. 8.10, 8.11)
- with *Quadratic*/*R**-variant (see Tab. 8.12)

Possible causes (hypotheses). Uniform distributions or distributions with very low noise generate weak start clusters. k-Means applied to such a distribution causes very strong overlap in all directions, as k-Means generates spherical clusters. This means that preprocessing is of little use because the MBRs overlap significantly.

Furthermore, quadratic and R*-Tree variants use reinsert and split heuristics. When we insert pre-clustered batches of points, these heuristics can trigger long split chains and move entries between nodes, which increases MBR overlap. This means that preprocessing is of little use because the tree recreates overlap and reduces the benefit of the groups.

At $d = 5$ and $d = 10$, overlap between MBRs is high enough to make kNN harder. At very high d, almost everything overlaps for both indexes, so the difference between strategies tends to level out. Our pre-clustering still helps in many cases, but the effect becomes more stable rather than growing without bound.

Risks in measurement. We assume queries come from the same distribution as the data ($q \sim \mathcal{P}$). This can be too optimistic. If the real query distribution shifts, the gains will likely drop.
Furthermore, "Nodes visited" is only a good metric for runtime in this test scenario. It ignores caching, memory/I/O, and implementation details, so it only approximates true execution time.

8.7 Summary

RQ1 (Query efficiency). Across 1080 test cases, the optimized index I^{opt} reduces node visits in most cases. The median relative gain is $\approx 8.3\%$ and the mean is $\approx 7.7\%$. About 60–65% of the cases are positive (see ECDF). There are a few strong negative outliers (down to -230%), and some strong wins (up to $+80\%$). Which was also shown in PLATON [YC23]. Efficiency depends heavily on the dataset. The experiments of this thesis confirm this finding. While improvements are achieved in most cases, there exist outliers with performance losses.

RQ2 (When does it help?). The most important factors are the *R-Tree variant*, *points per distribution*, and *dimensionality*. Simple rules from the decision tree: (i) **Linear**: if points per distribution > 2250, I^{opt} is usually better; (ii) **Quadratic**: if $2250 <$ points per distribution ≤ 4500, I^{opt} is often better; (iii) **R***: only useful in niche settings (e.g., high d, many distributions, higher noise). Gaussian data benefits more than uniform; very low noise and mid dimensions ($d = 5, 10$) are risk zones.

RQ3 (Build time). STR bulk loading is consistently faster for pure index construction. For 200k–500k points we observe about 8–13× faster builds with STR, even as d grows. In the optimized pipeline, *k-Means* and the C++ index build dominate the total build time; *split* and *BBox* are minor.

9 Summary and Outlook

In this final chapter, we summarize the key findings of this thesis, discuss its limitations, and outline potential directions for future work.

9.1 Key Findings

The optimization procedure presented in this thesis follows the tradition of using clustering for R-Tree constructions [BPT02], but goes beyond it by taking strict capacity limits into account for direct mapping of clusters to data pages (leaves). This thesis has shown that clustering methods, combined with an optimization algorithm, can significantly improve the efficiency of R-Tree index partitioning. By first applying k-Means clustering and then refining the results with a size-constraints-aware post-processing algorithm, we were able to create partitions that satisfy leaf constraints and reduce bounding box overlap. The experimental evaluation over 1080 test cases demonstrated that the optimized clustering approach improves query performance in the majority of scenarios, with a median relative gain of approximately 8%. Particularly in two and very high dimensions, and when using the linear split heuristic, the optimization achieved great improvements.

9.2 Limitations

Despite the overall positive results, there are some negative cases. In some parameter environments, especially when it comes to uniform distributions, medium-dimensional data (d=5 or d=10) and advanced R-Tree variants such as R*, the optimization provided only marginal or even negative improvements. This shows that the method is not universally beneficial for every use case and its effectiveness depends on data distribution, dimensionality, and index configuration. Also, the approach was only tested on synthetic datasets. Performance on real-world data may differ.

9.3 Future Work

Testing It is worth testing the work of this thesis on more data. We have already looked at 1080 test cases, but there are more possibilities to evaluate the usefulness. For example, it could be useful to simply expand the synthetic test data sets to test very high dimensionalities. The highest dimension is currently 20, but it might be worthwhile to test dimensions from 100 to 500 and above, as these dimensions are particularly relevant in current topics such as machine learning.

Bulk-Loading It should also be checked whether the algorithm can somehow be combined with bulk loading strategies of the R-Tree. Currently, the two mechanisms are not compatible with each other.

Intelligent Procedure Based on the tests and derived learned rules, an intelligent procedure could be developed that automatically decides whether an optimized index or a baseline index should be built on the input dataset. Recent work such as PLATON [YC23] shows that learning-based partitioning strategies offer further potential. Combining our approach with such methods could be an interesting field for future research.

Bibliography

[Arg+08] Lars Arge et al. "The Priority R-Tree: A Practically Efficient and Worst-Case-Optimal R-Tree". In: *ACM Transactions on Algorithms* 4 (Mar. 2008). DOI: 10.1145/1328911.1328920. URL: https://dl.acm.org/doi/10.1145/1328911.1328920.

[BBD00] P. S. Bradley, K. P. Bennett, and Ayhan Demiriz. *Constrained k-Means Clustering*. Tech. rep. MSR-TR-2000-65. Microsoft Research, May 2000. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2000-65.pdf.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. "The X-Tree: An Index Structure for High-Dimensional Data". In: *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, 1996, pp. 28–39. URL: https://bib.dbvis.de/uploadedFiles/58.pdf.

[BPT02] Sotiris Brakatsoulas, Dieter Pfoser, and Yannis Theodoridis. "Revisiting R-Tree Construction Principles". In: *Advances in Databases and Information Systems*. Ed. by Yannis Manolopoulos and Pavol Návrat. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 149–162. ISBN: 978-3-540-45710-7. DOI: 10.1007/3-540-45710-0_13. URL: https://www.dieter.pfoser.org/files/pubs/pdfs/brakatsoulas_rtree02.pdf.

[CH67] T. Cover and P. Hart. "Nearest neighbor pattern classification". In: *IEEE Transactions on Information Theory* 13.1 (1967), pp. 21–27. DOI: 10.1109/TIT.1967.1053964.

[Gut84] Antonin Guttman. "R-trees: a dynamic index structure for spatial searching". In: *SIGMOD Rec.* 14.2 (June 1984), pp. 47–57. ISSN: 0163-5808. DOI: 10.1145/971697.602266. URL: https://doi.org/10.1145/971697.602266.

[KF94] 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)*. Morgan Kaufmann, 1994, pp. 500–509. URL: https://www.vldb.org/conf/1994/P500.PDF.

[LLE97] Scott T. Leutenegger, Mario A. López, and Jeffrey Edgington. "STR: A Simple and Efficient Algorithm for R-Tree Packing". In: *Proceedings of the 13th IEEE International Conference on Data Engineering (ICDE)*. 1997. DOI: 10.1109/ICDE.1997.582015. URL: https://apps.dtic.mil/sti/tr/pdf/ADA324493.pdf.

[Llo82] S. Lloyd. “Least squares quantization in PCM”. In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: `10.1109/TIT.1982.1056489`.

[Mac67] J. MacQueen. “Some methods for classification and analysis of multivariate observations”. In: 1967. URL: `https://api.semanticscholar.org/CorpusID:6278891`.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. “Nearest neighbor queries”. In: vol. 24. 2. New York, NY, USA: Association for Computing Machinery, May 1995, pp. 71–79. DOI: `10.1145/568271.223794`. URL: `https://doi.org/10.1145/568271.223794`.

[YC23] Jieming Yang and Gao Cong. “PLATON: Top-down R-tree Packing with Learned Partition Policy”. In: *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data*. 2023. DOI: `10.1145/3626742`. URL: `https://dl.acm.org/doi/10.1145/3626742`.