COLLEGE OF COMPUTING - UM6P

School of Computer Science

Analysis of k-d Trees and R-Trees

Project D**: An experimental evaluation of high-dimensional tree-based vector indexing

Prepared by:
Hajar BELMOUDDEN

Teaching Assistant:
Khaoula ABDENOURI

Course Professor:
Karima ECHIHABI

December 29, 2024

Contents

List of Algorithms

List of Tables

List of Figures

1 Introduction

In the realm of data structures and spatial indexing, efficient organization and retrieval of multidimensional data have always been paramount challenges. Structures like k-d trees and R-trees are at the forefront of addressing these challenges by providing powerful partitioning methods for high-dimensional and spatial datasets. These data structures have become indispensable tools in various fields, including machine learning, geographic information systems (GIS), computer graphics, and database systems.

This document delves into the foundational concepts, definitions, and algorithms underlying k-d trees and their variants, including Squarish, Relaxed, and Median k-d trees. It explores their insertion and search procedures, formal definitions, and construction complexities. Additionally, the document extends the discussion to R-trees, a robust data structure for indexing spatial objects, and their optimized variants, such as Quadratic R-trees, R*-trees, and Revised R*-trees. Each variant is analyzed in terms of its efficiency, construction strategies, and pruning conditions, providing a comprehensive understanding of their respective strengths and trade-offs.

To evaluate the practical implications of these data structures, experimental results are presented, focusing on clustering quality, query efficiency, and construction performance. These insights offer a comparative analysis of the discussed data structures, equipping readers with a deeper understanding of their applications and limitations in real-world scenarios.

2 k-d Trees

A *kdtree* is a widely used data structure for partitioning space and storing points in a k-dimensional space. It is simple to understand and implement, and offers ease of updates and maintenance. This structure is particularly useful for searches that involve multidimensional keys. Additionally, k-d trees are suitable for various queries, including orthogonal range searches, partial match queries, and nearest neighbor searches.

In this chapter, we introduce the standard k-d tree data structure. In subsequent chapters, we will explore several variants of k-d trees, which differ from the standard version primarily in the insertion procedure, particularly in how one of the k dimensions is selected to split the search space. However, the same types of queries can be performed on all these variants of the kd tree.

2.1 Standard k-d Trees Definition

The k-d tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node, and points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k dimensions, with the hyperplane perpendicular to that dimension's axis.

For example, if the "x" axis is chosen for a particular split, all points in the subtree with a smaller "x" value than the node will appear in the left subtree, and all points with a larger "x" value will appear in the right subtree. In such a case, the hyperplane would be set by the x value of the point, and its normal would be the unit x-axis.

2.2 Formal Definition

A standard k-d tree for a set of k-dimensional keys is a binary tree where:

1. Each node contains a key $\mathbf{x} \in \mathbb{R}^k$, where $\mathbf{x} = (x[0], x[1], \ldots, x[k-1])$ and an associated discriminant $i \in \{0, \ldots, k-1\}$.
2. For every node with key $\mathbf{x}$ and discriminant i, any key $\mathbf{y}$ in its left subtree satisfies $y[i] < x[i]$, and any key $\mathbf{y}$ in the right subtree satisfies $y[i] \geq x[i]$.
3. The root node has depth 0 and discriminant 0. All nodes at depth d have discriminant $d \mod k$.

The space partitioning can be represented as:

$$\textbf{LeftSubtree} = \left\{\mathbf{y} \in \mathbb{R}^k : y[i] < x[i]\right\}, \quad \textbf{RightSubtree} = \left\{\mathbf{y} \in \mathbb{R}^k : y[i] \geq x[i]\right\}.$$

2.3 Insertion and Searching

The insertion and search operations in kd-trees, like binary search trees, use a specific coordinate (discriminant) at each recursion level to guide traversal. Stored at each node, the discriminant determines the comparison dimension. Both operations follow a **Depth-First Search (DFS)** pattern, exploring one branch deeply before backtracking, ensuring efficient navigation.

Algorithm 1 describes how to insert an element with key k and value v into the kd-tree T. It is assumed that the kd-tree does not already contain an element with key k before the insertion. If duplicates were to be allowed, the algorithm could be modified to include a condition $x = \text{key}$, and in this case, the old value associated with x would be updated to v.

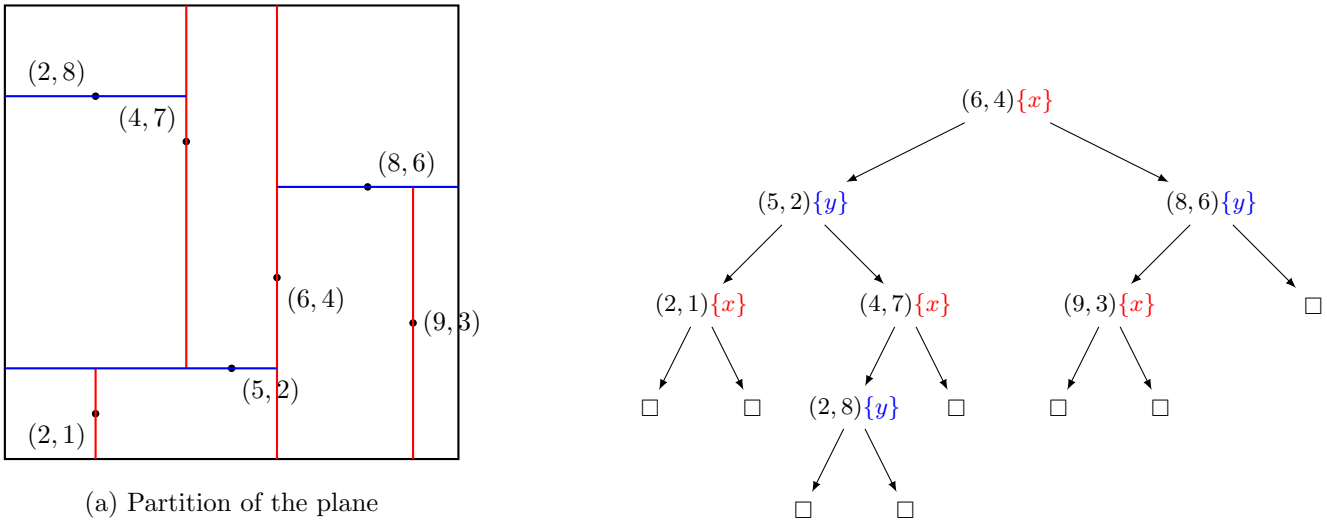
`InsertElement`(x, v) inserts an element with key k and value v in the current leaf. For standard kd-trees, the discriminant should be chosen as the parent's discriminant plus 1 (modulo k). This detail, which may depend on the specific kd-tree variant, is not explicitly included in the generalized algorithm.

Algorithm 1 Insertion in KD-Trees

```
1:  function INSERT(T, x, v)
2:      if T = ∅ then
3:          INSERTELEMENT(x, v)
4:      else
5:          key ← T.key;    i ← T.discr
6:          if x[i] < key[i] then
7:              INSERT(T.left, x, v)
8:          else
9:              INSERT(T.right, x, v)
10:         end if
11:     end if
12: end function
```

Figure 1 shows the kd-tree obtained after inserting $(6,4)$, $(5,2)$, $(4,7)$, $(8,6)$, $(2,1)$, $(9,3)$, and $(2,8)$ in this order into an initially empty kd-tree. In the figure, the total region is $[0,10] \times [0,10]$, assuming that we know that the inserted points will always fall into this square.

The first cut, made by $(6,4)$, is vertical at 6. The second cut, made by $(5,2)$, is horizontal at 2 but only affects the $[0,6) \times [0,10]$ subregion. The third cut, made by $(4,7)$, is vertical at 4, and only affects the $[0,6) \times [2,10]$ subregion, etc.



(a) Partition of the plane

(a) Partition of the plane

(b) 2d-tree structure

Figure 1: Partition of the plane and corresponding 2d-tree structure.

Note that every node represents both a point and a subregion of the plane, called the bounding box. For instance, with the assumption that all points fall into $[0,10] \times [0,10]$, the node with key $(4,7)$ represents this key and the bounding box $[0,6) \times [2,10]$.

On the other hand, if every node stored information about all the keys in its subtree, the bounding box of that node would shrink to the smallest rectangle enclosing all keys in the subtree. For example, it would become $[2,4] \times [7,8]$.

Another possible scenario arises when no information about future points or already-inserted points in the subtree is maintained. In this case, the bounding box of the node could be expanded, for instance, to $(-\infty, 6) \times [2, \infty)$. This bounding box can still be computed dynamically by traversing from the root to the node $(4,7)$.

The algorithms in my work assume **minimal information storage**, meaning nodes only store essential data like their key, value, and splitting dimension. This design keeps the tree lightweight and ensures compatibility across different scenarios. However, storing additional information, such as **bounding boxes** or **subtree data**, can enhance query performance by enabling more aggressive pruning during searches.

Algorithm 2 outlines the search procedure, which is similar to the insertion process. Here's the pseudocode :

Algorithm 2 Search in kd-trees.

```
1:  function SEARCH(T, x)
2:      if T = ∅ then
3:          return not found
4:      end if
5:      key ← T.key;    i ← T.discr
6:      if x = key then
7:          return found
8:      end if
9:      if x[i] < key[i] then
10:         return SEARCH(T.left, x)
11:     else
12:         return SEARCH(T.right, x)
13:     end if
14: end function
```

As an example, if we search for **(2, 8)** over the previous kd-tree, the key is found following the highlighted path.

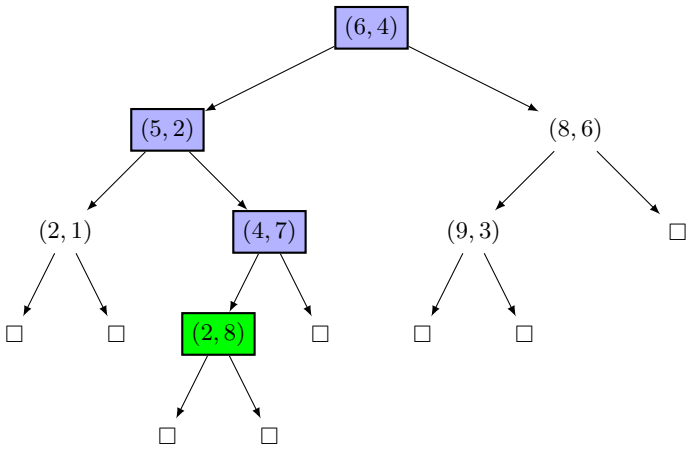


Figure 2: Search in a 2d-tree

3 Variants of k-d Trees

3.1 Squarish k-d Tree

3.1.1 Definition

This variant consists in a modification of the way the discriminant is chosen at every node. When a rectangle is split by a newly inserted point, instead of alternating the discriminants, the longest side of the rectangle is cut. Therefore, the cut is always a $(k-1)$-dimensional hyperplane through the new point as usual, but now it is always perpendicular to the longest edge of the rectangle (if there is a tie, the discriminant can be chosen at random). As a result, these k-d trees have more squarish-looking regions, which is why this variant was named squarish k-d trees.

Note that, compared to **Definition 2.2**of standard k-d trees, this variant implies changes only in the third condition, so as to reflect the new method to choose the discriminant.

3.1.2 Formal Definition

A squarish k-d tree for a set of k-dimensional keys is a binary tree where:

- Each node contains a key $x \in \mathbb{R}^k$, where $x = (x_0, x_1, \ldots, x_{k-1})$ and an associated discriminant $i \in \{0, \ldots, k-1\}$.
- For every node with key x and discriminant i, any key y in its left subtree satisfies $y[i] < x[i]$, and any key y in the right subtree satisfies $y[i] \geq x[i]$.
- Each node v is associated with a k-dimensional rectangular region $R(v) = [l_0, u_0] \times [l_1, u_1] \times \cdots \times [l_{k-1}, u_{k-1}]$.
- The discriminant i for a node v is chosen as:

$$i = \arg \max_{0 \leq j < k} (u_j - l_j)$$

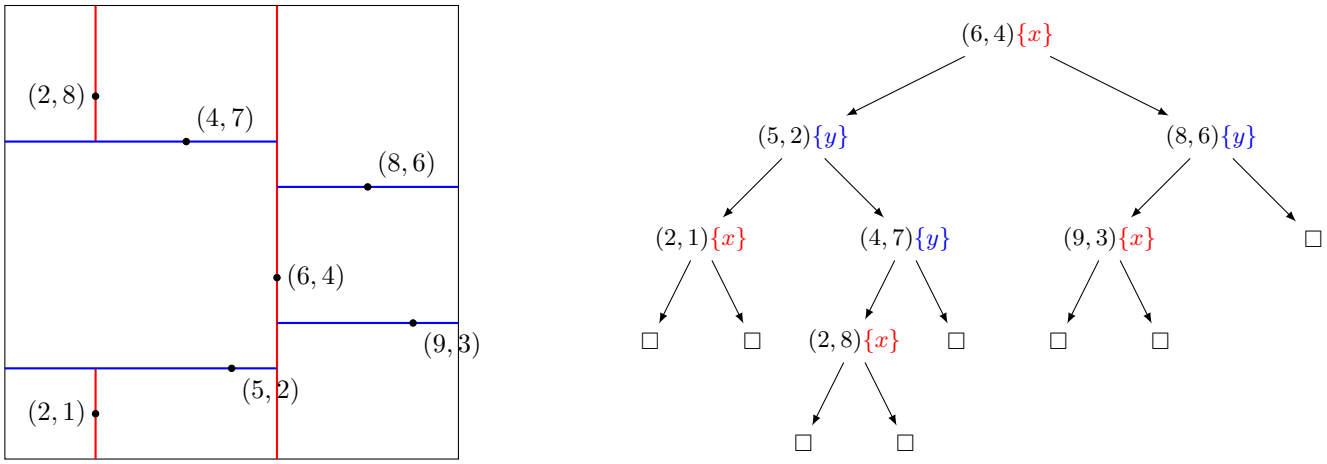
  If there is a tie, i can be chosen randomly among the dimensions with the longest edge.

- The space partitioning for a node v with key x and discriminant i can be represented as:

$$\text{LeftSubtree} = \{y \in R(v) : y[i] < x[i]\}$$

$$\text{RightSubtree} = \{y \in R(v) : y[i] \geq x[i]\}$$

- For a child node v' of v, its associated region $R(v')$ is updated as follows:
  - If v' is a left child:
$$R(v') = R(v) \cap \{y \in \mathbb{R}^k : y[i] < x[i]\}$$
  - If v' is a right child:
$$R(v') = R(v) \cap \{y \in \mathbb{R}^k : y[i] \geq x[i]\}$$



(a) Partition of the plane

(b) 2d-tree structure

Figure 3: Partition of the plane and corresponding 2d-tree structure.

3.1.3 Example

3.2 Relaxed k-d Tree

3.2.1 Definition

The relaxed k-d tree is a variant of the standard k-d tree where discriminants are chosen randomly. When a new node is inserted, a random dimension between 0 and $k-1$ is selected, independent of previous decisions, tree structure, or the inserted point.
Compared to **Definition 2.2**, the relaxed k-d tree omits the third condition, which requires a structured discriminant choice.

3.2.2 Formal Definition

A relaxed k-d tree for a set of k-dimensional keys is a binary tree where:

- Each node contains a k-dimensional record and has an associated arbitrary discriminant $j \in \{0, 1, \ldots, k-1\}$.
- For every node with key x and discriminant j, any record in the left subtree with key y satisfies $y[j] < x[j]$, and any record in the right subtree with key y satisfies $y[j] \geq x[j]$.

3.3 Median k-d Tree

A Median KD-Tree is a spatial partitioning data structure that improves upon the standard KD-Tree by adaptively selecting splitting dimensions to create more balanced partitions. Unlike traditional KD-Trees that cycle through dimensions, the Median KD-Tree chooses the splitting dimension that best approximates the median of the space at each node, resulting in a more balanced tree structure and improved search efficiency.

This variant is particularly useful for applications requiring frequent nearest neighbor searches or range queries, as it helps maintain consistent performance by minimizing the occurrence of unbalanced subtrees.

3.3.1 Formal Definition

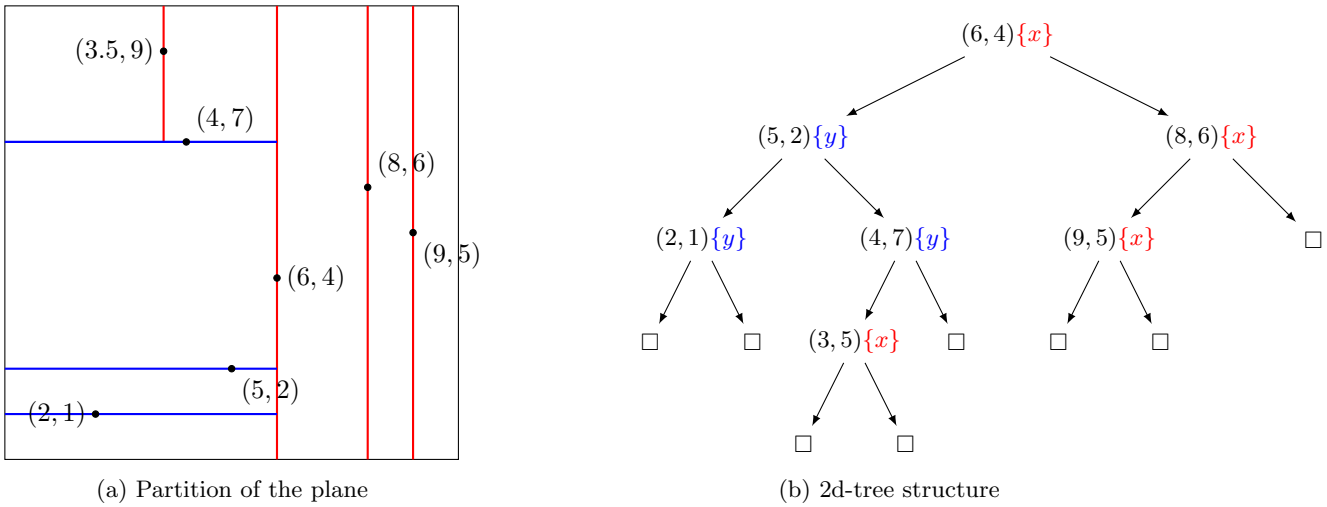
Note that every leaf of a K-d tree corresponds to a region of the space from which the elements are drawn, and hence the whole tree induces a partition of the space, $[0,1]^K$ in our case. The region delimited by the leaf that a new node replaces at the moment of its insertion into the tree is known as its *bounding box*.

In median K-d trees, when a new data point $\mathbf{x} = (x_0, x_1, \ldots, x_{K-1})$ is inserted into the bounding box $\mathbf{R} = [\ell_0, u_0] \times \cdots \times [\ell_{K-1}, u_{K-1}]$, the discriminant j is chosen as follows:

$$j = \arg \min_{0 \le i < K} \left(\frac{x_i - \ell_i}{u_i - \ell_i} - \frac{1}{2} \right)$$

3.3.2 Example

An example of a median K-d tree, together with its induced partition of the space, is shown in Fig. 4



(a) Partition of the plane

(b) 2d-tree structure

Figure 4: Example of a median K-d tree built from 2-dimensional points.

4 Distance Metrics and Pruning Condition in k-d Trees

Given a query point Q in a k-dimensional space, a nearest neighbor candidate C, and a splitting hyperplane S at a node in the k-d tree, the distance between Q and C can be calculated using different distance metrics. While the **pruning condition** remains fundamentally the same across all k-d tree variants, the method for selecting the splitting dimension differs.

1. Distance Metrics

The distance between Q and C can be calculated using the following metrics:

(a) Euclidean Distance (L2 norm)

The Euclidean distance measures the straight-line distance between Q and C:

$$d(Q, C) = \sqrt{\sum_{i=1}^{k} (q_i - c_i)^2}$$

The distance from Q to the splitting hyperplane S is:

$$d(Q, S) = |q_{\text{split}} - s|$$

where q_{split} is the coordinate of Q along the splitting dimension, and s is the value of the splitting hyperplane.

(b) Manhattan Distance (L1 norm)

The Manhattan distance measures the grid-based distance between Q and C:

$$d(Q, C) = \sum_{i=1}^{k} |q_i - c_i|$$

The distance to the splitting hyperplane remains:

$$d(Q, S) = |q_{\text{split}} - s|$$

(c) Chebyshev Distance (L norm)

The Chebyshev distance measures the largest single coordinate difference between Q and C:

$$d(Q, C) = \max_{i=1}^{k} |q_i - c_i|$$

Again, the distance to the splitting hyperplane is:

$$d(Q, S) = |q_{\text{split}} - s|$$

2. Pruning Condition

The pruning condition for k-d trees is:

$$d(Q, S) \geq d(Q, C)$$

Where:

- $d(Q, C)$ is the distance between the query point Q and the current nearest neighbor candidate C.

- $d(Q, S)$ is the distance from Q to the splitting hyperplane S.

If the pruning condition is satisfied:

- The subregion on the far side of the splitting hyperplane can be **pruned**, as it cannot contain a point closer than the current candidate C.

If the pruning condition is not satisfied:

- The far side of the split must be explored, as it may contain a closer point.

3. Key Differences in k-d Tree Variants

While the pruning condition remains consistent, the method for selecting the splitting dimension (i) varies across k-d tree variants:

(a) Standard k-d Trees

The splitting dimension cycles through the available dimensions.

(b) Squarish k-d Trees

The splitting dimension is chosen as the longest edge of the current region:

$$i = \arg \max_{0 \leq j < k} (u_j - l_j)$$

Where $[l_j, u_j]$ represents the extent of the rectangular region along dimension j.

(c) Median k-d Trees

The splitting dimension is chosen to best approximate the median of the space.

(d) Relaxed k-d Trees

The splitting dimension j is chosen randomly for each node during tree construction, independent of the tree structure or inserted points. The distance to the splitting hyperplane is calculated as:

$$d(Q, S) = |q_j - x_j|$$

Where j is the randomly chosen splitting dimension for the current node.

4. Summary

1. **Pruning Condition:**

   $$d(Q, S) \geq d(Q, C)$$

   This condition is identical across all k-d tree variants and ensures efficient pruning of the search space.

2. **Splitting Dimension Selection:**

- **Standard k-d trees:** Cyclic selection.
- **Squarish k-d trees:** Based on the longest edge.
- **Median k-d trees:** Approximate median.
- **Relaxed k-d trees:** Random selection.

3. **Distance Metrics:** The distance between points and to the splitting hyperplane can be computed using Euclidean, Manhattan, or Chebyshev distances, depending on the application.

The relaxed k-d tree's random selection strategy can lead to more balanced trees on average, improving performance for certain data distributions.

5 Construction Complexity Analysis of the k-d Trees

5.1 Standard k-d Trees

The construction complexity involves recursively splitting a dataset into two halves based on a discriminant at each node.

1. At each node, the dataset is split along a selected discriminant axis, dividing the data into left and right subtrees.
2. The splitting involves scanning all points to determine which ones fall on the left and right of the hyperplane, which takes $O(n)$ time for n points.
3. The process repeats recursively for each half until all points are inserted into the tree.

Depth of Recursion :

- In a balanced KD-tree, the recursion depth is proportional to $\log n$, where n is the number of points.
- Each recursion level processes all remaining points.

Mathematical Proof :
At the i-th level of recursion:

- The dataset size is halved: $n/2^i$.
- Processing all points at level i takes $O(n/2^i)$ time.

The total time $T(n)$ for the tree construction is the sum of the time taken across all levels:

$$T(n) = \sum_{i=0}^{\log n} O\left(\frac{n}{2^i}\right)$$

This is a geometric series:

$$T(n) = n \sum_{i=0}^{\log n} \frac{1}{2^i}$$

The sum of this series is:

$$T(n) = n \cdot O(1) = O(n \log n)$$

Thus, constructing a standard KD-tree takes $O(n \log n)$.

5.2 Squarish k-d Trees

Steps in Construction :

1. At each node, the discriminant is chosen as the dimension with the largest extent in the bounding box:
$$i = \arg \max_{0 \le j < k} (u_j - l_j)$$
Computing the discriminant takes $O(k)$, where k is the dimensionality.
2. Points are partitioned into two subsets: those on the left and right of the splitting hyperplane. Partitioning all points takes $O(n)$ time for n points at the current level.
3. The process is recursively applied to the left and right subsets.

Depth of Recursion :
For a balanced squarish k-d tree:

- The depth is $O(\log n)$, as each split divides the dataset approximately in half.

At the i-th level:

- Number of points processed is $\frac{n}{2^i}$.
- Time to determine the splitting dimension for all points:
$$O\left(\frac{n}{2^i} \cdot k\right)$$
- Partitioning points takes:
$$O\left(\frac{n}{2^i}\right)$$

Total Time Complexity :

Summing over all levels of recursion:

$$T(n) = \sum_{i=0}^{\log n} \left[O\left(\frac{n}{2^i} \cdot k\right) + O\left(\frac{n}{2^i}\right)\right]$$

Simplifying:

$$T(n) = O(kn) \sum_{i=0}^{\log n} \frac{1}{2^i}$$

The geometric series $\sum_{i=0}^{\log n} \frac{1}{2^i}$ converges to 2, so:

$$T(n) = O(kn \log n)$$

Thus, the construction complexity of a squarish k-d tree is:

$$O(kn \log n)$$

5.3 Relaxed k-d Trees

Steps in Construction :

1. For each node:
   - Choose a random discriminant $j \in \{0, 1, \ldots, k-1\}$.
   - Partition points into left and right subtrees based on the selected discriminant j.
   - Continue recursively for the left and right subtrees.
2. Since the discriminant is chosen randomly:
   - No specific balancing strategy is enforced.
   - Tree depth and shape depend on the randomness of the splits.

Depth of Recursion :

In expectation:

- A random split divides the dataset into two approximately equal parts.
- The depth of the tree is proportional to $\log n$, where n is the number of points.

However, due to randomness:

- There is a possibility of highly unbalanced splits, leading to a depth of $O(n)$ in the worst case.

Total Time Complexity :

For each level of recursion:

- Partitioning n points into two subsets takes $O(n)$.
- Randomly selecting a discriminant takes $O(1)$.

For a balanced tree (average case):

- There are $O(\log n)$ levels, and the work at each level sums to:

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \ldots = O(n)$$

Thus, the total complexity is:

$$T_{\text{construction}} = O(kn \log n)$$

For the worst case (highly unbalanced tree):

- Depth can reach $O(n)$, leading to:

$$T_{\text{construction-worst}} = O(kn^2)$$

5.4 Median k-d Trees

Key Steps in Construction :

1. **Choosing the Splitting Dimension**: At each node, the splitting dimension j is chosen adaptively as:
$$j = \arg\min_{0 \le i < k} \left| \frac{x_i - \ell_i}{u_i - \ell_i} - \frac{1}{2} \right|$$
This involves evaluating the normalized distance of the median point x_i from the center $\frac{1}{2}$ for each dimension. Computing this for k dimensions requires $O(k)$ time per node.

2. **Finding the Median**: After selecting the splitting dimension, the data points are sorted along that dimension to find the median. Using the *Median of Medians* algorithm, the median can be found in $O(n)$ time for n points.

3. **Splitting the Data**: The dataset is split into two halves (left and right subtrees) based on the median. This requires $O(n)$ time to partition the points.

4. **Recursive Calls**: The same process is recursively applied to the two halves of the dataset.

Mathematical Analysis :

Let $T(n)$ denote the time complexity for constructing a Median k-d Tree for n points. At each level:

- The splitting dimension is determined in $O(k)$.
- The median is found in $O(n)$ using the *Median of Medians* algorithm.
- The dataset is split in $O(n)$.
- Two recursive calls are made, each for $n/2$ points.

Recurrence Relation :

$$T(n) = O(k) + O(n) + 2T\left(\frac{n}{2}\right)$$

Solving the Recurrence :

Step 1: Ignore the Constant Work at Each Node. The constant work per level is $O(k+n) = O(n)$, assuming k is fixed and much smaller than n.

Step 2: Expand the Recurrence. Expand the recurrence relation over $\log n$ levels (height of a balanced tree):

$$T(n) = O(n) + 2T\left(\frac{n}{2}\right)$$

$$T(n) = O(n) + 2\left[O\left(\frac{n}{2}\right) + 2T\left(\frac{n}{4}\right)\right]$$

$$T(n) = O(n) + O(n) + 4T\left(\frac{n}{4}\right)$$

Step 3: Generalize the Pattern. At level i, there are 2^i subproblems, each with size $n/2^i$. The cost of work at level i is:

$$\text{Cost at level } i = 2^i \cdot O\left(\frac{n}{2^i}\right) = O(n)$$

Step 4: Sum Over All Levels. There are $\log n$ levels in a balanced tree. Summing over all levels:

$$T(n) = \sum_{i=0}^{\log n} O(n) = O(n \log n)$$

Thus, the construction complexity of a **Median k-d Tree** is:

$$T_{\text{construction}} = O(n \log n)$$

Summary of complexities :

KDTree Variant	Construction Complexity
Standard KDTree	$O(n \log n)$
Squarish KDTree	$O(kn \log n)$
Relaxed KDTree	$O(kn \log n)$ (Best/Average), $O(kn^2)$ (Worst)
Median KDTree	$O(n \log n)$

Table 1: Construction Complexities for KDTree Variants

6 R Trees

6.1 R Trees Definition

An **R-tree** is a widely used data structure for indexing multi-dimensional spatial data. It is efficient for organizing and querying spatial information, offering good performance for various spatial queries. This structure is particularly useful for applications involving: Geographic Information Systems (GIS), Computer-Aided Design (CAD) and multi-dimensional databases.

R-trees are suitable for various spatial queries, including:

- **Range Searches**: Finding all spatial objects within a specified range.
- **Nearest Neighbor Searches**: Locating the closest spatial object(s) to a query point.

In this chapter, we introduce the **standard R-tree** data structure. In subsequent chapters, we will explore several variants of R-trees, which differ from the standard version primarily in how they organize and split nodes,they minimize overlap and how they maximize space utilization. However, the same types of spatial queries can be performed on all these variants of the R-tree.

The R-tree organizes spatial objects into a hierarchy of **Minimum Bounding Rectangles (MBRs)**, allowing for efficient pruning of the search space during queries. Unlike **k-d trees** which partition space into non-overlapping regions, R-trees allow for overlapping regions, which can

provide more flexibility in organizing spatial data but may also lead to increased query complexity in some cases.

An R-tree structures spatial data as a hierarchy of *Minimum Bounding Boxes (MBBs)*, where the lowest level contains the *data objects*, and higher levels consist of MBBs enclosing those below.

Definition (Minimum bounding box). Given a spatial object X, the minimum bounding box $M = \text{MBB}(X)$ is such that:

- M is an axis-aligned box,
- $X \subseteq M$,
- No M' satisfying conditions 1 and 2 exists such that $|M'| < |M|$,

where $|M|$ represents the volume of the bounding box M.

For a k-dimensional space, the MBB can be represented as:

$$\text{MBB}(X) = [l_1, u_1] \times [l_2, u_2] \times \cdots \times [l_k, u_k]$$

where:

$$l_i = \min(x_i \mid x \in X) \quad \text{(lower bound in dimension } i\text{)},$$
$$u_i = \max(x_i \mid x \in X) \quad \text{(upper bound in dimension } i\text{)}.$$

This definition ensures that the MBB is the smallest axis-aligned box that fully contains the spatial object X.

Definition (R-tree entry). An entry in the R-tree is either an R-tree node or a data object.

Definition (Data object). A data object is a spatial object that belongs to a specific data set D within a given data space Ω. More formally:

$$\text{A spatial object } X \text{ is considered a data object if and only if } X \in D,$$

where D is a set of spatial objects such that:

$$\forall X \in D : X \subseteq \Omega.$$

In simpler terms, a data object is any spatial entity that:

- Is part of the defined data set D,
- Resides entirely within the boundaries of the data space Ω,
- Can be searched and retrieved during query operations.

Data objects form the basic units of information that are indexed and queried in spatial data structures like k-d trees and R-trees.

Definition (R-tree node). An R-tree node N comprises a collection of entries as described previusly . For any node that is not the root, it must adhere to the following condition:

$$m \leq |N| \leq M,$$

where m denotes the minimum number of entries required in a node (minimum fill grade), and M represents the maximum number of entries a node can hold (maximum node size).

Definition (Minimum Bounding Box of a Node). The Minimum Bounding Box (MBB) of an R-tree node N, denoted as MBB(N), is defined recursively as the smallest bounding box that contains the MBBs of all its child entries. Formally:

$$\text{MBB}(N) = \text{MBB}\left(\bigcup_{E \in N} \text{MBB}(E)\right).$$

Although an entry does not explicitly store its MBB, it can be efficiently generated from the tree's structure, functioning as a cached value. This makes retrieving the MBB a low-cost operation in practice, despite being omitted in theoretical discussions for simplicity.

6.2 Example

Below is provided an example structure of an R-tree where each node has 2 children. The data objects are illustrated at the bottom denoted X1 through X4, and M1 through M6 are the MBBs .

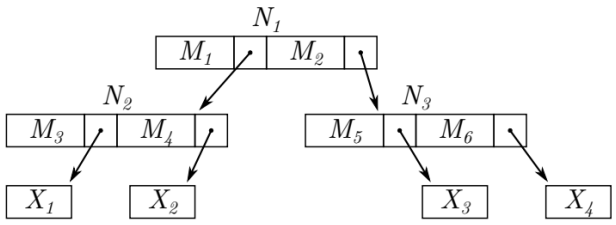


Figure 5: R tree example

6.3 Insertion and Searching

Range Search

The range search operation in R-trees involves traversing the hierarchical structure of the tree while eliminating nodes whose bounding boxes (MBBs) do not overlap with the query object Q, as described in the Algorithm3 . This approach leverages the property that all child nodes of a parent node N are contained within the MBB of N. Consequently, this property allows the search process to efficiently exclude irrelevant nodes, making it significantly faster than a brute-force linear scan in many practical cases.

Unlike range searches in other structures such as B-trees, range queries in R-trees may explore the entire tree in the worst case. The computational complexity of a range search is therefore linear in the number of nodes in such cases. However, the practical efficiency of the search is heavily influenced by the strategy used for grouping entries into nodes during the tree's construction.

Algorithm 3 Range search in R-tree.

```
1: function RangeSearch(N, Q)
2:     for all E ∈ N do
3:         if MBB(E) ∩ Q = ∅ then
4:             continue
5:         end if
6:         if E is a node then
7:             RangeSearch(E, Q)
8:         else
9:             report E
10:        end if
11:    end for
12: end function
```

As described in Algorithm 3, `RangeSearch(`N, Q`)` involves traversing the R-tree by iterating through the entries of a node and checking for intersections with the query box.

Note: This algorithm employs a **Depth-First Search (DFS)** strategy, as it recursively explores each branch of the tree before moving on to the next entry in the current node.

6.3.1 Insertion

The process of inserting objects into an R-tree follows a general procedure, as outlined in Algorithm 4. The algorithm begins by navigating the tree to locate an appropriate node for the new data object. Once the correct position is identified, the new object is added, and the changes are propagated upwards through the tree, with nodes potentially splitting as needed.

It is worth noting that in Algorithm 4, the input O represents an entry rather than a direct data object. This distinction becomes more evident when discussing variants like the R*-tree. For the initial call, O is given as a pair (M, p), where M is the bounding box (MBB) of the data object, and p is a pointer to the data object itself. The parameter N always starts as the root node.

The method `ChooseSubtree` is responsible for selecting a child node within N that determines which subtree the new entry should be placed into. This step ultimately identifies the leaf node where the insertion will occur. If a node becomes overfull after the addition, the `SplitNode` method is invoked. This method splits the entries of the node into two groups, and the resulting nodes replace the original node in its parent.

If the algorithm returns a pair of nodes where the second node is not empty, the root node is replaced by a new node containing both elements of the pair. This mechanism allows the tree to grow in height when necessary. Otherwise, the algorithm updates the tree with the returned node as the root, completing the insertion process.

Algorithm 4 Insertion algorithm for dynamic construction of an R-tree, where N is a node and O is an entry. `ChooseSubtree` and `SplitNode` depend on the implementation.

```
1: function INSERT(O, N)
2:     if O belongs at the level of N then
3:         add O to N
4:     else
5:         E ← ChooseSubtree(N, O)
6:         (N, N') ← Insert(O, E)
7:         if N' ≠ ∅ then
8:             add N' to N
9:         end if
10:    end if
11:    if N is overfull then
12:        return SplitNode(N)
13:    end if
14:    return (N, ∅)
15: end function
```

7 R-tree Variants

As noted in the previous Section, the performance of an R-tree is strongly influenced by how data objects are grouped into nodes, with worst-case complexity being linear to the number of nodes.

This section explores algorithms for R-tree construction, focusing solely on dynamic methods that allow incremental insertions, as deletions are assumed to be infrequent. Specific variants discussed include the Quadratic R-tree, R*-tree, and Revised R*-tree.

Constructing an R-tree often involves optimization, requiring a method to compare elements based on a function's value. Additionally, many algorithms need a tie-breaking mechanism, which can be simplified using a partial order over tuples.

Definition (Partial Order on Tuples):
Given two tuples $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$, $A \leq B$ if there exists an index i where $a_i < b_i$, and all previous elements are equal ($a_j = b_j$ for all $j < i$).
More formally : $A \leq B \iff \exists i \in [1, n] : a_i < b_i \wedge \forall j \in [1, i], a_j = b_j$

This concept is useful when minimizing a function f with a tie-breaker function g. By defining $f'(x) = (f(x), g(x))$, the problem becomes:

$$Y =_{x \in D} f'(x)$$

where $Y \subseteq D$ is the set of values minimizing f with g as the tie-breaker.

7.1 Quadratic R-tree

A **Quadratic R-tree** is a hierarchical spatial data structure designed to optimize node splitting and reduce the overlap and total volume of Minimum Bounding Boxes (MBBs). This directly improves query efficiency by minimizing the number of intersecting nodes.

1. Structure

Each node in the tree contains up to m entries, where m is the node capacity. Each entry consists of:

- An MBB M that encloses the spatial extent of its child entries (for internal nodes) or a single data object (for leaf nodes).
- A pointer p, which points to either child nodes (internal entries) or the actual data object (leaf entries).

2. Splitting Strategy (`SplitNode Algorithm`)

The splitting process divides an overfull node N into two groups G_1 and G_2 to minimize overlap and wasted space. Formally:

1. **Seed Selection:**
   - Let $N = \{E_1, E_2, \ldots, E_m, E_{m+1}\}$ represent the entries in the overfull node.
   - Select two entries E_i, E_j that maximize wasted space:
   $$W(E_i, E_j) = \text{vol}(M_i \cup M_j) - \text{vol}(M_i) - \text{vol}(M_j)$$
   - Initialize $G_1 = \{E_i\}$ and $G_2 = \{E_j\}$.
2. **Assign Remaining Entries:** For each remaining entry $E \in N \setminus \{E_i, E_j\}$, compute the volume enlargement for adding E to G_1 and G_2:
   $$\Delta_{G_k}(E) = \text{vol}(M_{G_k} \cup M_E) - \text{vol}(M_{G_k}), \quad k \in \{1, 2\}$$
   Assign E to the group with the smallest $\Delta_{G_k}(E)$. In case of a tie:
   (a) Assign to the group with the smallest volume ($\text{vol}(M_{G_k})$).
   (b) If still tied, assign to the group with fewer entries.

3. Insertion Strategy (`ChooseSubtree Algorithm`)

When inserting an object O, the subtree whose MBB requires the least volume enlargement is selected:

$$N' =_{E \in N} \Delta_O \text{vol}(M_E)$$

where:

$$\Delta_O \text{vol}(M_E) = \text{vol}(M_E \cup M_O) - \text{vol}(M_E)$$

In case of a tie, the subtree with the smallest $\text{vol}(M_E)$ is chosen.

4. Volume Definition

The volume of an MBB $M = (b, t)$ in a d-dimensional space is:

$$\text{vol}(M) = \prod_{i=1}^{d} (t[i] - b[i])$$

where $b[i]$ and $t[i]$ are the lower and upper bounds of M along the i-th dimension.

5. Optimization Objectives

The Quadratic R-tree aims to:

1. Minimize the total volume of MBBs:

$$\text{minimize} \sum_{N \in \text{Tree}} \text{vol}(M_N)$$

2. Minimize overlap between MBBs:

$$\text{minimize} \sum_{(M_i, M_j) \in \text{Tree}} \text{vol}(M_i \cap M_j)$$

7.2 R*-Tree

The **R*-Tree** is an enhanced variant of the R-tree, introduced by Beckmann and Seeger. It incorporates reinsertion, improved split strategies, and additional metrics like perimeter and overlap of Minimum Bounding Boxes (MBBs). These features aim to optimize query performance while adapting the tree to varying data distributions.

1. Key Features of R*-Tree

1. **Reinsertion:**
   - Instead of always splitting overfull nodes, the R*-tree reinserts a subset of entries into higher levels.
   - The p entries farthest from the node center are chosen for reinsertion, where the center distance is:

$$d_c(M_1, M_2) = \sqrt{\sum_{i=1}^{d} \left(\frac{b_{1,i} + t_{1,i}}{2} - \frac{b_{2,i} + t_{2,i}}{2} \right)^2}$$

     Here, M_1 and M_2 are MBBs, and b, t are their bounds.

2. **Perimeter and Overlap:**
   - The **perimeter** of an MBB B is:

$$\text{perim}(B) = \sum_{i=1}^{d} (t[i] - b[i])$$

   - The **overlap** of B with a set of boxes X is:

$$\Omega_X(B) = \sum_{B' \in X} \text{vol}(B \cap B')$$

3. **Adaptation to Data Distribution:** Reinsertion allows the tree to adapt to changing data distributions by redistributing entries inserted early when the data distribution was not fully known.

2. Insertion Strategy (`ChooseSubtree Algorithm`)

When inserting an object, the R*-tree selects a subtree using a multi-step process:

1. If O fits within any child's MBB, select the child with the smallest volume.
2. For internal nodes, minimize volume enlargement:

$$\Delta_O \text{vol}(E) = \text{vol}(M_E \cup M_O) - \text{vol}(M_E)$$

3. For leaf nodes, minimize overlap enlargement:

$$\Delta_O \Omega_N(E) = \Omega_N(M_E \cup M_O) - \Omega_N(M_E)$$

   If ties occur, selection is based on volume or perimeter. To reduce computational overhead, only the p-best candidates (e.g., $p = 32$) are considered.

3. Node Splitting Strategy (`SplitNode Algorithm`)

When a node becomes overfull, the split strategy partitions entries to minimize overlap, volume, and perimeter:

1. Identify split candidates (S_1, S_2) satisfying:

$$S_1 \cup S_2 = S, \quad S_1 \cap S_2 = \emptyset, \quad m \leq |S_1|, |S_2| \leq M$$

   where m and M are the minimum and maximum node fill levels.

2. For each dimension i:
   - Sort entries by their i-th dimension bounds.
   - Partition into two groups and calculate the perimeter:

$$\text{perim}(S_1) + \text{perim}(S_2)$$

   - Choose the dimension minimizing this sum.

3. Select the candidate minimizing overlap:

$$\text{vol}(S_1 \cap S_2)$$

4. Optimization Objectives

The R*-tree aims to:

1. Minimize overlap and volume:

$$\text{minimize} \sum_{(M_i, M_j) \in \text{Tree}} \text{vol}(M_i \cap M_j), \quad \text{minimize} \sum_{N \in \text{Tree}} \text{vol}(M_N)$$

2. Reduce query cost by limiting the number of intersecting nodes.

7.3 Revised R*-Tree

The **Revised R*-Tree** is an enhanced version of the R*-Tree, which introduces new optimizations such as the removal of reinsertion, skewed splits to adapt to data distribution, and perimeter-based metrics for better handling of MBBs with no volume.

1. Perimeter and Overlap

- The **perimeter** of an MBB $B = (b, t)$ is:

$$\text{perim}(B) = \sum_{i=1}^{d} (t[i] - b[i])$$

- The **overlap** of B with a set of boxes X is:

$$\Omega_X(B) = \sum_{B' \in X} \text{perim}(B \cap B')$$

2. MBB Change Recording

Each node N stores its MBB at the time of creation ($\text{MBB}_0(N)$) and its current MBB ($\text{MBB}(N)$). These are used to calculate an **asymmetry score**:

$$a_i = \frac{(t[i]_1 + b[i]_1) - (t[i]_0 + b[i]_0)}{t[i]_1 - b[i]_1}$$

Here, (b_1, t_1) and (b_0, t_0) are the bounds of $\text{MBB}(N)$ and $\text{MBB}_0(N)$, respectively.

3. Skewed Splits

Skewed splits allocate more space in directions where data is expected to expand. The **split index** adjusts the asymmetry score using:

$$\mu_i = \left(1 - \frac{2m}{M+1}\right) a_i$$

This index is used in conjunction with a Gaussian-based function $w_f(C)$ to evaluate split candidates.

4. Insertion Strategy (`ChooseSubtree Algorithm`)

When inserting a new object, the Revised R*-Tree selects the subtree that minimizes perimeter and overlap enlargement:

1. If the object O is contained within any child MBB, select:

$$N' =_{E \in S} \begin{cases} \text{perim}(E), & \text{if vol}(E) = 0 \\ \text{vol}(E), & \text{otherwise} \end{cases}$$

   where $S = \{E \in N : O \in \text{MBB}(E)\}$.

2. If no child contains O:

   - Minimize perimeter enlargement $\Delta_O \text{perim}(E)$.
   - For children with non-zero overlap, select the child minimizing:

$$\Delta_O \Omega_N \text{perim}(E)$$

5. Node Splitting Strategy (`SplitNode Algorithm`)

The Revised R*-Tree splits nodes by considering lower and upper bounds of entries:

1. For each dimension i, calculate split candidates:

$$P_i(S) = L_i(S) \cup U_i(S)$$

   where $L_i(S)$ and $U_i(S)$ are splits sorted by lower and upper bounds, respectively.

2. Select the split candidate minimizing a **goal function**:

$$w(S_1, S_2) = \begin{cases} \frac{w_g(S_1,S_2)}{w_f(S_1,S_2)}, & w_g(S_1, S_2) \geq 0 \\ w_g(S_1, S_2) \cdot w_f(S_1, S_2), & w_g(S_1, S_2) < 0 \end{cases}$$

3. The goal function $w_g(S_1, S_2)$ uses perimeter for MBBs with no volume:

$$w_g(S_1, S_2) = \begin{cases} \text{perim}(S_1) + \text{perim}(S_2) - p_{\max}, & \text{if MBB}(S_1) \cap \text{MBB}(S_2) = \emptyset \\ f(\text{MBB}(S_1) \cap \text{MBB}(S_2)), & \text{otherwise} \end{cases}$$

   where $f = \text{perim}$ if any candidate lacks volume, and $f = \text{vol}$ otherwise.

4. The function $w_f(C)$ adjusts the split based on data distribution:

$$w_f(C) = y_s \exp\left(-\frac{(x(C) - \mu_i)^2}{s(1 + |\mu_i|)}\right) - y_1$$

   where:

$$y_1 = \exp\left(-\frac{1}{s^2}\right), \quad y_s = \frac{1}{1 - y_1}, \quad x(C) = \frac{2|C_1|}{M + 1} - 1$$

7.4 Pruning Conditions for R-trees and Their Variants

The pruning condition for R-trees and their variants is a fundamental aspect of their efficiency in spatial querying. This condition allows the search algorithm to eliminate irrelevant branches of the tree, significantly reducing the search space and improving query performance.

1. Standard R-tree Pruning Condition

For the standard R-tree, the pruning condition during a range search is formally defined as:

$$\text{MBB}(N) \cap Q = \emptyset$$

where:

- $\text{MBB}(N)$ is the Minimum Bounding Box of node N,
- Q is the query object (typically a range or region).

This condition states that if the intersection between the MBB of a node and the query object is empty, the entire subtree rooted at that node can be pruned from the search.

2. Pruning in R-tree Variants

The pruning condition remains fundamentally the same for R-tree variants such as the Quadratic R-tree, R*-tree, and Revised R*-tree. However, these variants introduce optimizations that indirectly affect pruning efficiency:

Quadratic R-tree

- **Objective:** Minimize the total volume and overlap of MBBs.
- **Pruning Benefit:** Reduced overlap leads to fewer intersections with the query object, allowing more effective pruning.

R*-tree

- **Objective:** Introduce reinsertion and improved split strategies while incorporating additional metrics like perimeter and overlap.
- **Pruning Benefit:** Better-organized MBBs result in more efficient pruning during queries.

Revised R*-tree

- **Objective:** Introduce skewed splits and perimeter-based metrics.
- **Pruning Benefit:** Adapts better to data distribution, potentially leading to more effective pruning in certain scenarios.

3. Mathematical Explanation

The effectiveness of pruning in R-trees is closely related to the concept of MBB overlap. Consider two nodes A and B with their respective MBBs:

$$\text{MBB}(A) = [l_1^A, u_1^A] \times [l_2^A, u_2^A] \times \cdots \times [l_k^A, u_k^A]$$
$$\text{MBB}(B) = [l_1^B, u_1^B] \times [l_2^B, u_2^B] \times \cdots \times [l_k^B, u_k^B]$$

The overlap between these MBBs is defined as:

$$\text{vol}(\text{MBB}(A) \cap \text{MBB}(B)) = \prod_{i=1}^{k} \max\left(0, \min(u_i^A, u_i^B) - \max(l_i^A, l_i^B)\right)$$

Minimizing this overlap is crucial for efficient pruning. The R-tree variants achieve this through different strategies:

- **Quadratic R-tree:** Minimizes wasted space during node splits.
- **R*-tree:** Uses a combination of volume and overlap metrics in its `ChooseSubtree` and `SplitNode` algorithms.
- **Revised R*-tree:** Introduces perimeter-based overlap calculations and skewed splits to better handle data distribution.

By reducing overlap, these variants increase the likelihood of the pruning condition being met during queries, thus improving overall query performance.

8 Construction Complexity Analysis of R-Trees and Variants

1. Standard R-Tree

Key Steps in Construction:

- **Insertion:** At each level, the `ChooseSubtree` algorithm selects the most suitable subtree by minimizing the increase in the volume of the Minimum Bounding Box (MBB). This requires $O(m)$ comparisons, where m is the branching factor ($m \leq M$).
- **Splitting:** When a node overflows, the `SplitNode` algorithm divides the entries into two groups. For standard R-trees:
  - Evaluating all pairs of entries takes $O(M^2)$ time.
  - Assigning the remaining entries to groups requires $O(M)$ time.

Complexity Analysis:

1. **Height of the Tree:** For n entries with maximum node capacity M, the height of the tree is:
$$h = O(\log_M n)$$
2. **Insertion Cost:** At each level, $O(M)$ comparisons are required to select the appropriate subtree.
3. **Splitting Cost:** Splitting a node costs $O(M^2)$.

4. **Total Construction Cost:** With n entries, each insertion requires traversal of $h = O(\log_M n)$ levels. Occasionally, a split occurs. The total cost is:

$$T_{\text{construction}} = O(n \cdot \log_M n \cdot M^2)$$

Simplifying, since M is constant:

$$T_{\text{construction}} = O(n \log n)$$

2. Quadratic R-Tree

Key Features:

- **Seed Selection:** Choose two entries E_i and E_j that maximize the wasted space:

$$W(E_i, E_j) = \text{vol}(M_i \cup M_j) - \text{vol}(M_i) - \text{vol}(M_j)$$

This step requires $O(M^2)$ comparisons for $M + 1$ entries.

- **Entry Assignment:** Each remaining entry is assigned to a group, requiring $O(M)$ computations per entry.

Splitting Cost:

$$T_{\text{splitting}} = O(M^2)$$

Total Construction Cost:

$$T_{\text{construction}} = O(n \cdot \log_M n \cdot M^2) = O(n \log n)$$

3. R*-Tree

Key Features:

- **Reinsertion:** A subset of entries (p) is reinserted when a node overflows. Reinsertion requires sorting entries by their distance from the center:

$$d_c(M_1, M_2) = \sqrt{\sum_{i=1}^{d} \left(\frac{b_{1,i} + t_{1,i}}{2} - \frac{b_{2,i} + t_{2,i}}{2} \right)^2}$$

Reinsertion cost is $O(p \log p)$.

- **Split Strategy:** For each dimension, entries are sorted ($O(M \log M)$) and partitioned. The total cost is:

$$T_{\text{splitting}} = O(k \cdot M \log M)$$

Total Construction Cost:

$$T_{\text{construction}} = O(n \cdot \log_M n \cdot M^2) + O(n \cdot k \cdot \log M) + O(n \log M)$$

Simplifying:

$$T_{\text{construction}} = O(n \log n)$$

4. Revised R*-Tree

Key Features:

- Introduces skewed splits and perimeter-based metrics, reducing reinsertion overhead.
- Skewed splits evaluate asymmetry scores and goal functions, costing $O(k \cdot M \log M)$ per split.

Total Construction Cost:

$$T_{\text{construction}} = O(n \cdot \log_M n \cdot k \cdot M \log M) = O(n \log n)$$

Summary of Construction Complexities

R-Tree Variant	Construction Complexity
Standard R-Tree	$O(n \log n)$
Quadratic R-Tree	$O(n \log n)$
R*-Tree	$O(n \log n)$
Revised R*-Tree	$O(n \log n)$

Table 2: Construction Complexities for R-Tree Variants

9 Experimental Results and Analysis

In this chapter, I present and discuss the results of the experiments conducted to evaluate the behavior and performance of the KD-tree variants explored in this work. These experiments serve to compare the newly implemented variants with existing ones especially in time of construction .

9.1 Overview of Experiments

I carried out a series of experiments on all implemented KD-tree variants, testing their performance across a high-dimensional dataset. The variants include some well-known types: **Standard KD-tree**, **Squarish KD-tree**, and **Relaxed KD-tree**, as well as newly introduced variants: **Median KD-tree**.

The experiments focused on several key aspects:

- **Construction Time**: Measuring the time required to build each tree.
- **Clustering Quality**: Evaluating the quality of the partitions formed by the trees using clustering metrics.
- **Neighbor Search Efficiency**: Assessing how effectively the trees retrieve nearest neighbors for given test points.

The experiments were conducted using the **GIST dataset**, consisting of:

- **1 million training points** in a 960-dimensional feature space for tree construction.

- **1,000 test queries** to evaluate neighbor search performance.
- **Ground truth neighbors** for validation purposes.

For clustering quality assessment, I used three well-established metrics:

1. Kaliński-Harabasz Score
2. Davies-Bouldin Score
3. Silhouette Score

9.2 Evaluation of the Standard KD-Tree

The **Standard KD-tree** was implemented as a recursive binary tree, designed to partition data points in a high-dimensional space. Each node in the tree represents a k-dimensional point and splits the dataset along a chosen axis, determined cyclically by the depth of the tree. Points with values lower than the split on the chosen axis are assigned to the left subtree, while the rest are assigned to the right.

The implementation supports the following core functionalities:

1. **Insertion**: Points are inserted into the tree based on their position relative to the current node's discriminant axis. The process continues recursively until a suitable leaf node is found.
2. **Search**: Points can be searched efficiently by traversing the tree, narrowing down possibilities based on the discriminant axis at each level.
3. **Cluster Formation**: Clusters are identified by grouping points in subtrees that meet a minimum depth threshold, leveraging the recursive structure of the tree to aggregate data based on spatial proximity.

To evaluate this implementation, I measured key performance metrics, including:

- **Index Construction Time**: The time required to build the tree with 1 million points.
- **Clustering Quality**: The grouping of points into meaningful clusters, assessed using clustering metrics.
- **Querying Speed**: The efficiency of nearest-neighbor searches performed on 1,000 test queries.

9.2.1 Visualization of clusters :

At first , I begin with providing a 3D scatter plot that depicts clusters identified by the KD-tree algorithm after grouping points in the dataset based on a minimum depth threshold of 2.

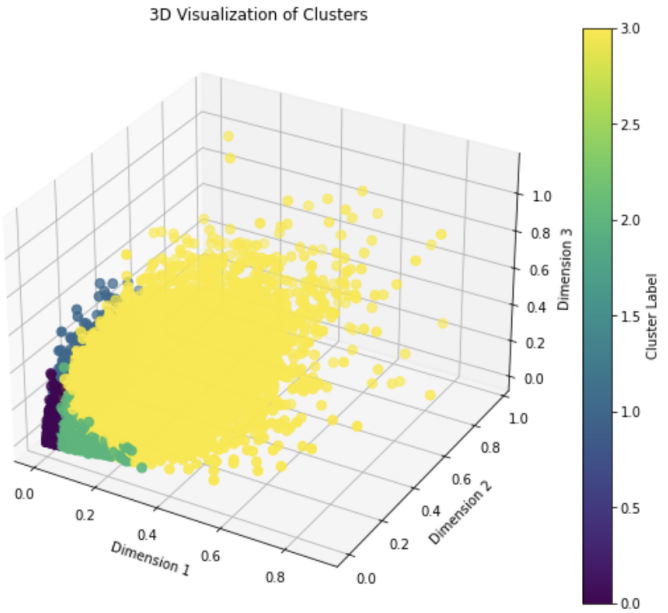


Figure 6: 3D Scatter Plot of Clusters Identified by the Standard KD-Tree.

Here's a detailed analysis of the results:
Cluster Overview :

- The plot represents four distinct clusters, as evidenced by the `Clusters formed: 4` output.
- Each cluster is color-coded for easy identification, with the color gradient ranging from dark (smaller cluster labels) to light (higher cluster labels).

Cluster Characteristics :

1. **Cluster 0**:
    - Contains 118,506 points.
    - Represented by darker tones in the plot.
    - Likely occupies a more compact or well-defined region in the feature space due to its smaller size.
2. **Cluster 1**:
    - Contains 276,441 points.
    - Intermediate size with moderately dispersed points across the space.
    - Appears to overlap slightly with other clusters at its edges, which may indicate proximity to transitional regions between clusters.

3. **Cluster 2**:

    - Contains 5,733 points, making it the smallest cluster.
    - Located in a distinct and possibly sparse region of the feature space.
    - Its isolation in the plot suggests a highly specific partition, likely due to the recursive nature of the KD-tree splits.

4. **Cluster 3**:

    - The largest cluster with 599,317 points.
    - Dominates the plot with a broader spread, indicating that it encompasses a significant portion of the feature space.
    - The lighter coloration suggests higher variability within this cluster.

Spatial Arrangement

- The three axes (Dimension 1, Dimension 2, Dimension 3) represent reduced dimensions or direct projections of the high-dimensional dataset.
- Clusters show varying densities and spreads, indicating that the KD-tree partitioning captured regions of differing data distributions.

Insights :

1. **Cluster Balance:**

    - There is a noticeable imbalance in cluster sizes, with Cluster 3 dominating the dataset and Cluster 2 being relatively small. This reflects the hierarchical splitting nature of the KD-tree, which prioritizes geometric partitioning without ensuring uniform cluster sizes.

2. **Overlapping Regions:**

    - Some points near the boundaries of clusters appear closer to points from other clusters. This could either represent natural overlap in the data distribution or limitations of the KD-tree's axis-aligned splits in capturing diagonal or curved boundaries.

3. **Data Structure Representation:**

    - The clustering aligns well with the KD-tree's logic, as seen by the relatively distinct partitions. However, the differences in size and spread highlight areas where additional techniques could enhance the results.

Conclusion : This visualization effectively demonstrates the Standard kd-tree's ability to partition high-dimensional data into distinct clusters. The large variation in cluster sizes and densities suggests that the dataset exhibits significant structural complexity.

9.2.2 Evaluation of the Quality Clustering :

The bar plot, as you can below, represents the evaluation of clustering quality for the KD-tree using three metrics: **Silhouette Score**, **Calinski-Harabasz Index**, and **Davies-Bouldin Index**. These metrics provide a comprehensive perspective on how well the KD-tree partitions the dataset.

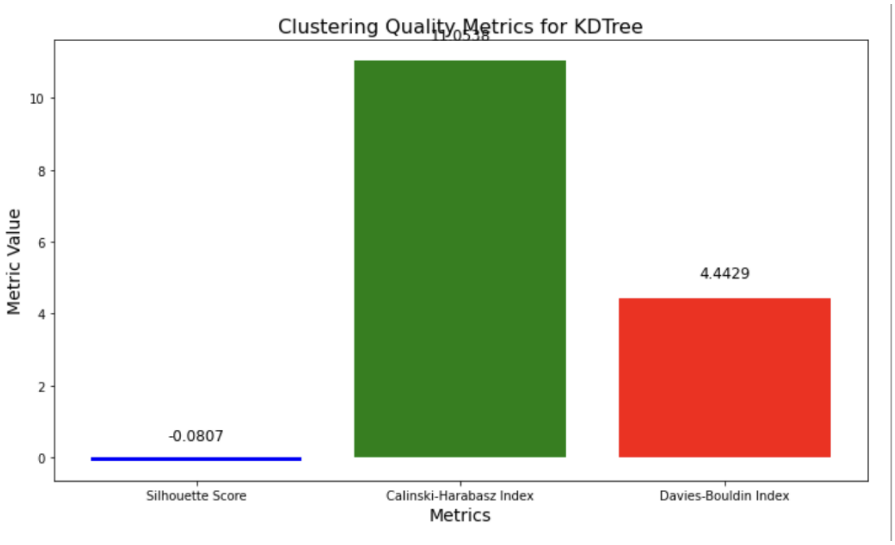


Figure 7: Clustering Quality Metrics Plot for the Standard KD-Tree.

I have used several metics to evlauate the quality of clusetring :

1. **Silhouette Score**:
   - Measures how well-separated clusters are by calculating the difference between intra-cluster cohesion and inter-cluster separation.
   - **Range:** $[-1, 1]$.
   - **Interpretation:**
     - A value near **1** indicates well-separated and compact clusters.
     - A value near **0** indicates overlapping clusters.
     - A **negative value** (-0.0807) suggests points may be assigned to incorrect clusters or clusters are poorly separated.
2. **Calinski-Harabasz Index**:
   - Measures the ratio of inter-cluster dispersion (separation) to intra-cluster dispersion (compactness).
   - **Higher values** indicate better-defined clusters.

- The value of **11.0518** here suggests that clusters are reasonably compact and well-separated, indicating good overall clustering quality.

3. **Davies-Bouldin Index**:

   - Measures the average similarity between each cluster and its most similar cluster.
   - **Lower values** indicate better clustering quality, reflecting higher separation and lower compactness issues.
   - The value of **4.4429** indicates moderate clustering quality, where clusters are defined but may still overlap or lack ideal compactness.

Interpretation of Results :

- **Silhouette Score**: The negative Silhouette Score highlights significant overlap between clusters or poor assignment of points, a limitation of KD-tree's axis-aligned splits in high dimensions.
- **Calinski-Harabasz Index**: The high value reflects that clusters are well-separated relative to their compactness, demonstrating the KD-tree's ability to partition the dataset effectively.
- **Davies-Bouldin Index**: The moderate value suggests that while clusters are functional, there is room for improvement in terms of inter-cluster separation and intra-cluster compactness.

Overall Analysis :
The KD-tree performs well in separating the dataset into clusters, as evidenced by the high Calinski-Harabasz Index. However, the low Silhouette Score indicates points near cluster boundaries are often misclassified. The moderate Davies-Bouldin Index highlights overlapping clusters and irregular boundaries, suggesting room for improvement.

As a conclusion to this analysis ,the clustering quality metrics reveal the strengths and limitations of the KD-tree:

- Clusters are generally well-separated, but boundary overlap and irregularities need optimization.
- Exploring non-axis-aligned splits or hybrid KD-tree variants could improve clustering quality further.

Now, after computing all the evaluation metrics for the Standard KD-tree, I am evaluating the clustering quality based on the **Silhouette Score** as a function of the number of dimensions.

The plot below illustrates how the Silhouette Score changes with increasing dimensionality:

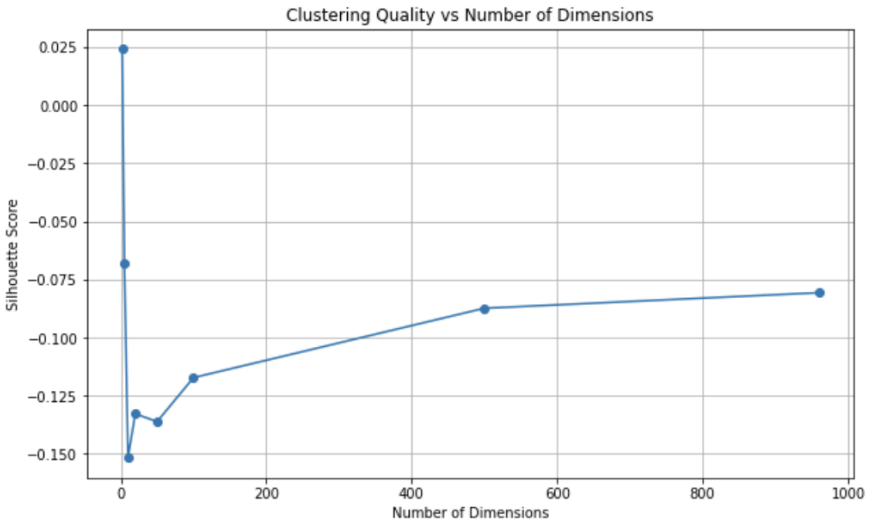


Figure 8: Clustering Quality based on the Silhouette score vs Number of dimensions.

1. **Initial Drop:**
    - At very low dimensions, the score starts slightly positive but rapidly drops into negative values.
    - This indicates a sharp decline in clustering quality due to the *curse of dimensionality*, which reduces the effectiveness of axis-aligned splits in forming meaningful clusters.
2. **Gradual Recovery:**
    - After reaching a minimum around 100-200 dimensions, the score begins to improve slightly.
    - Despite this recovery, the scores remain negative, reflecting persistent overlap or poorly separated clusters.
3. **Asymptotic Behavior:**
    - Beyond 600 dimensions, the score stabilizes and improves only marginally.
    - This plateau indicates that further increases in dimensionality have a minimal effect on clustering quality.

Key Insights :

- The consistently negative Silhouette Scores highlight the **limitations of axis-aligned KD-tree splits** in capturing meaningful clustering structures in high-dimensional spaces.

- The slight recovery at higher dimensions might be attributed to the increasing sparsity of data, which reduces overlap between clusters, albeit without forming well-separated clusters.

This analysis underscores the importance of considering dimensionality when evaluating the KD-tree's performance for clustering tasks.

9.2.3 Evaluation of the k-d Tree Build Time :

The plot as you can see below demonstrates a **linear relationship** between KDTree build time and data size, reflecting efficient scalability. The build time remains low for smaller datasets (< 25 MB) but increases proportionally with larger data, aligning with the expected $O(n \log n)$ complexity.

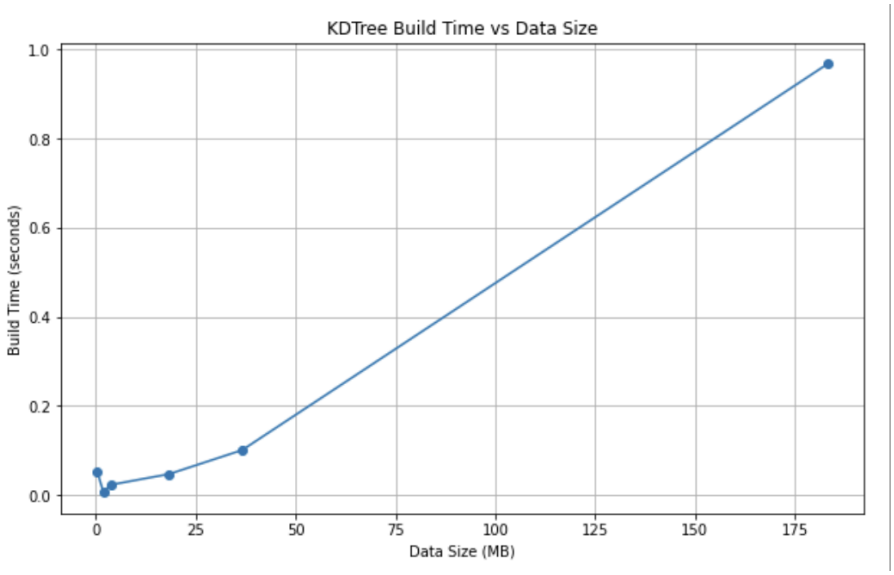


Figure 9: KDTree Build Time vs. Data Size Plot.

In fact, he implementation efficiently handles moderately large datasets (up to 175 MB) within 1 second, making it suitable for most applications. However, the consistent growth in build time highlights the impact of recursive splitting in high-dimensional data (960 features). While the KDTree scales well for current dataset sizes, it may face limitations with very large datasets (e.g., GB-sized), where optimizations like parallelization or dimensionality reduction could become necessary to maintain performance. This analysis confirms the KDTree's practicality for medium-sized datasets while pointing to potential enhancements for larger-scale applications.

9.2.4 Evaluation of the k-d Tree Query Speed vs Leaf Size :

Additionaly , I have plotted how query speed (queries per second) varies with the leaf size (depth threshold) in the KDTree. Key observations include:

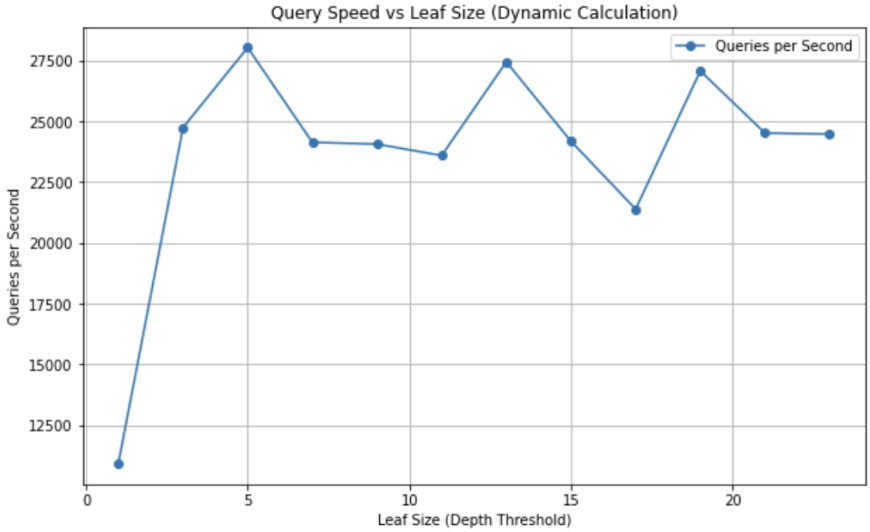


Figure 10: Query Speed vs Leaf Size Plot.

Performance Trends :

1. **Initial Increase:** Query speed rises sharply with smaller leaf sizes, peaking at an optimal threshold. This reflects reduced tree depth, minimizing traversal time.
2. **Fluctuations:** After the peak, query speeds stabilize with minor oscillations, showing the tree's adaptability to varying configurations.
3. **Decline at Larger Sizes:** Query speed decreases at larger leaf sizes due to fewer, broader clusters that reduce partition granularity.

Practical Insights :

- The **optimal leaf size** balances traversal depth and cluster precision, maximizing query performance.
- The KDTree maintains consistent efficiency across a range of thresholds, making it suitable for diverse real-time querying tasks.

This analysis highlights the importance of tuning leaf size to achieve the best trade-off between speed and partition precision.

9.2.5 Evaluation of the k-d Index Size vs Data Size :

Last but not least ,this plot demonstrates the relationship between the index size (in MB) and the data size (number of points) for the KDTree.

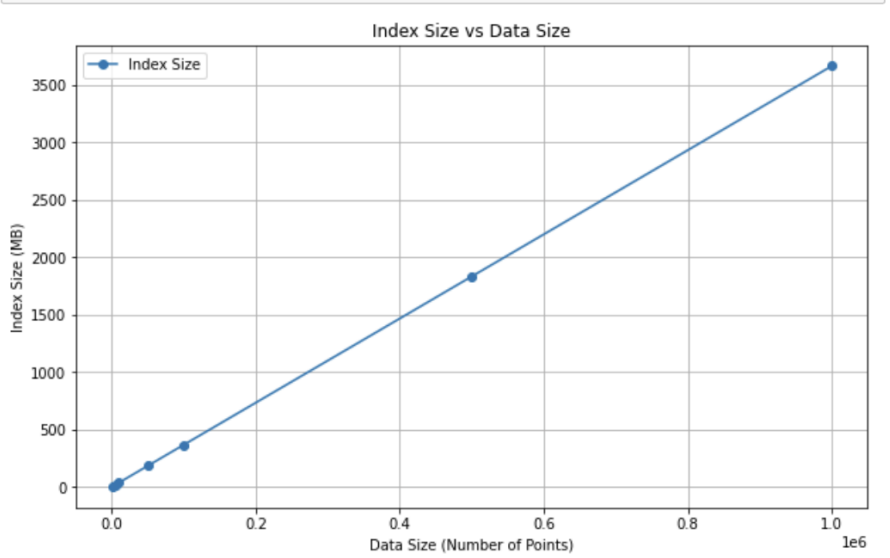


Figure 11: Index Size vs Data Size Plot.

The following key insights are drawn from this plot :

1. Linear Growth The index size increases linearly with the number of data points. This is expected, as the index size is directly proportional to the number of points and their dimensionality, given the consistent structure of the KDTree and fixed memory requirements per value (4 bytes for Float32).

2 . Impact of Dimensionality With 960 dimensions, each data point contributes significantly to the overall index size. For instance, a dataset with 1 million points results in an index size of approximately 3.67 GB, highlighting the memory-intensive nature of high-dimensional datasets.

3. Scalability The linear relationship indicates that the KDTree index is scalable with dataset size, maintaining a predictable growth in storage requirements. However, for extremely large datasets, storage optimization or dimensionality reduction may be necessary to keep the index size manageable.

4. Practical Implications

- **Memory Requirements:** This plot provides a clear understanding of the memory footprint of the KDTree. Applications with limited memory should assess dataset size and dimensionality before constructing the index.
- **Optimization Potential:** Techniques such as dimensionality reduction (e.g., PCA) or quantization could help reduce the index size without significantly affecting the KDTree's utility.

In summary, the plot highlights the strong linear relationship between data size and index size, emphasizing the importance of considering storage constraints when working with high-dimensional datasets. The predictable growth ensures scalability, but large datasets may require additional optimizations to remain efficient.

9.3 Interpretation of Tree Construction Times for KDTree Variants

Now moving to the comparison of construction times for KDTree variants, the bar plot highlights significant differences in the time required to build the different KDTree structures using the same training dataset.

Below are the observations and insights:

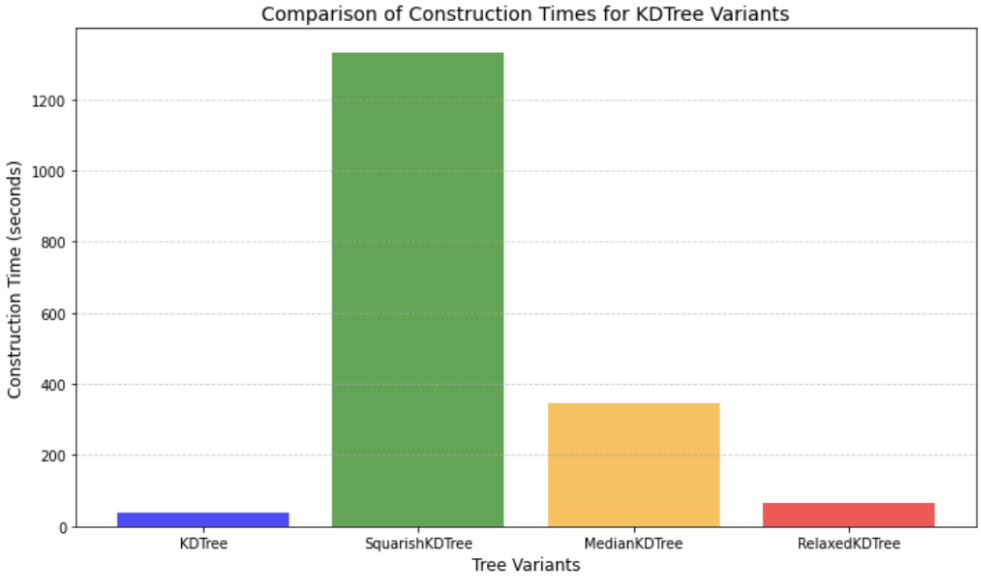


Figure 12: Comparison of construction times of kd tree variants.

Performance Comparison :

- **KDTree**:
  - The standard KDTree is the most efficient, with a construction time of **39.70 seconds**.
  - Its simplicity and median-based axis-aligned splits contribute to its low computational overhead, making it suitable for quick tree construction.
- **SquarishKDTree**:
  - The SquarishKDTree exhibits the longest construction time, taking **1333.26 seconds**.
  - This extended duration reflects the additional computational complexity involved in optimizing the aspect ratio of partitions during tree construction. While this may improve clustering quality, it comes at a significant cost in terms of efficiency.
- **MedianKDTree**:

- The MedianKDTree takes **344.49 seconds**, which is substantially longer than the standard KDTree but much faster than the SquarishKDTree.
- The enforced median splits across all dimensions add extra computation compared to the standard KDTree, but the impact is moderate compared to the Squarish variant.

- **RelaxedKDTree**:
  - The RelaxedKDTree construction time is **64.98 seconds**, slightly longer than the standard KDTree but far more efficient than SquarishKDTree and MedianKDTree.
  - Its relaxed splitting strategy, which allows slight imbalances, achieves a balance between construction speed and clustering effectiveness.

Key Insights :

- **Trade-Offs:**
  - The results highlight a clear trade-off between construction efficiency and the computational complexity of tree-building strategies.
  - SquarishKDTree sacrifices construction speed for potentially better clustering performance, while RelaxedKDTree and MedianKDTree strike a middle ground.

- **Scalability:**
  - For applications requiring frequent tree reconstruction or very large datasets, the standard KDTree or RelaxedKDTree are preferable due to their faster construction times.
  - The SquarishKDTree, due to its significant overhead, may only be viable for static datasets where clustering quality is prioritized.

In conclusion , this comparison underlines the importance of choosing the right KDTree variant based on the application's specific needs. While SquarishKDTree focuses on improving clustering quality at a high computational cost, the standard KDTree and RelaxedKDTree emphasize efficiency, making them better suited for real-time or large-scale scenarios.

10 Conclusion and challenges

To conclude my work , this project hve explored the theoretical and practical aspects of k-d trees and R-trees, including their well-known and enhanced variants. These data structures have demonstrated their effectiveness in multidimensional data organization and querying, with unique trade-offs in construction time, accuracy, and efficiency.

While the Standard k-d Tree offers simplicity and speed, advanced variants such as the Squarish k-d Tree and R*-Tree prioritize clustering quality and spatial indexing efficiency. The experimental evaluations highlighted the practical differences among these structures, showcasing how specific configurations cater to distinct application needs.

Despite the challenges faced, this project provides valuable insights into the behavior of k-d trees and R-trees under real-world constraints. It also sets the foundation for future research into optimizing these data structures for specific tasks and domains.

Challenges Encountered

During my work on this project, I faced several obstacles that influenced the outcomes and scope of the research:

1. **Time Constraints:** The most significant challenge was the limited time available to conduct comprehensive research. Due to this constraint, I could not delve deeply into all aspects of multidimensional data structures, such as metric trees, which were omitted from this study.

2. **Experimental Data Challenges:** The dataset used for experiments was extremely large, making it difficult to process effectively within the available computational resources. To manage this, I had to slice the dataset, which resulted in reduced accuracy and limited the evaluation of clustering quality and query performance.

3. **Implementation Complexity:** Implementing advanced tree variants like the Squarish k-d Tree and Revised R*-Tree was computationally intensive and required significant effort. This complexity limited my ability to perform exhaustive testing across different configurations and parameters.

4. **Resource Limitations:** Both memory and computational power were constrained, making it challenging to handle large-scale datasets and experiments efficiently. This limitation affected the ability to evaluate certain tree variants on their scalability and performance with larger datasets.

These obstacles reflect the practical difficulties of working on such a research project independently, especially within constrained time and resources. Overcoming these challenges would require additional computational support, extended timelines, and collaborative efforts to deepen the exploration of these data structures. Despite these hurdles, this research provides a solid contribution to understanding and applying k-d trees and R-trees in multidimensional data indexing.

11 References

- http://delab.csd.auth.gr/books/rtrees/chapter2.pdf
- http://www.gitta.info/SpatPartitio/en/html/ObjOriDecomp_learningObject2.html
- https://www.researchgate.net/publication/2452855_Visualizing_and_Animating_R-trees_and_Spatial_Operations_in_Spatial_Databases_on_the_Worldwide_Web
- https://www.ibm.com/docs/bg/informix-servers/14.10?topic=method-insertion-into-r-tree-index
- https://en.wikipedia.org/wiki/K-d_tree
- https://upcommons.upc.edu/bitstream/handle/2099.1/11309/MasterMercePons.pdf?sequence=1&isAllowed=y
- https://www.baeldung.com/cs/k-d-trees
- https://www.cs.upc.edu/~duch/home/duch/candamADB.pdf
- https://luc.devroye.org/jabbour.pdf
- https://en.wikipedia.org/wiki/R-tree
- https://en.wikipedia.org/wiki/R*-tree
- https://www.cs.upc.edu/~duch/home/duch/candamADB.pdf