Experimental Evaluation of High-Dimensional Tree-Based Vector Indexing

Adnane AIT MAGOURT

December 28, 2024

1 KD-Tree

1.1 KD-Tree Overview

A KD-Tree is a binary tree used to organize points in a **k-dimensional space**. Each internal node represents a **hyperplane** that splits the space into two subspaces, and this process continues recursively.

1.2 Discriminant Choice

The discriminant is the dimension (axis) along which the data is split at each node. Different strategies are used to select this dimension.

1.2.1 Alternating the Discriminant in a Cyclic Way

The discriminant dimension is selected by cycling through the dimensions in a fixed order. If the current depth is d, the discriminant is:

$$\text{discriminant}_d = D[(d-1) \mod k+1]$$

Algorithm 1 Alternate Dimension

Require: nbr_dims, $last\_dim$
Ensure: Next dimension to split on
1: $dim \leftarrow (last\_dim + 1) \mod nbr\_dims$
2: **return** $\{dim\}$

Time Complexity: $O(1)$
Space Complexity: $O(1)$

1.2.2 Randomly Choosing the Discriminant (Baseline)

The discriminant dimension is chosen uniformly at random from $\{1, 2, \ldots, k\}$:

$$\text{discriminant}_d = \text{rand}_d \in \{1, 2, \ldots, k\}$$

Algorithm 2 Random Dimension

Require: *nbr_dims*
Ensure: Random dimension to split on
1: $dim \leftarrow$ random integer between 0 and $nbr\_dims - 1$
2: **return** $\{dim\}$

Time Complexity: $O(1)$
Space Complexity: $O(1)$

1.2.3 Identifying the Direction of Maximum Variance

The dimension with the highest variance is chosen as the discriminant. The variance σ_i^2 along dimension i is:

$$\sigma_i^2 = \frac{1}{n} \sum_{j=1}^{n} (x_{j,i} - \mu_i)^2$$

Algorithm 3 Max Variance Dimension

Require: *datapoints*, *nbr_dims*
Ensure: Dimension with highest variance
1: $max\_variance \leftarrow 0$
2: $max\_variance\_dim \leftarrow 0$
3: **for** $dim = 0$ to $nbr\_dims - 1$ **do**
4: Compute mean of points in dimension dim
5: Compute variance for dimension dim
6: **if** variance > max_variance **then**
7: $max\_variance \leftarrow$ variance
8: $max\_variance\_dim \leftarrow dim$
9: **end if**
10: **end for**
11: **return** $\{max\_variance\_dim\}$

Time Complexity: $O(n \cdot k)$
Space Complexity: $O(1)$

1.2.4 The Discriminant is the Dimension of the Widest Interval

The dimension with the largest interval (difference between max and min values) is selected:

$$\text{Interval}_i = [\min(x_{1,i}, x_{2,i}, \ldots, x_{n,i}), \max(x_{1,i}, x_{2,i}, \ldots, x_{n,i})]$$

Algorithm 4 Widest Interval Dimension

Require: *datapoints*, *nbr_dims*
Ensure: Dimension with widest range
1: *max_range* ← 0
2: *max_range_dim* ← 0
3: **for** *dim* = 0 to *nbr_dims* − 1 **do**
4: Compute max and min values for dimension *dim*
5: *range* ← max - min
6: **if** *range* > *max_range* **then**
7: *max_range* ← range
8: *max_range_dim* ← *dim*
9: **end if**
10: **end for**
11: **return** {*max_range_dim*}

Time Complexity: $O(n \cdot k)$
Space Complexity: $O(1)$

1.3 Split Position Choice

Once the discriminant dimension is selected, the **split position** determines where to divide the data.

1.3.1 The Median Value of Our Data Points Across the Chosen Dimension

The median value is used as the split position:

$$\text{median}_d = \text{median}(x_1, x_2, \ldots, x_n)$$

Algorithm 5 Median Split

Require: *datapoints*, *dim*
Ensure: Median value along the chosen dimension
1: *sorted_points* ← Sort points along dimension *dim*
2: *mid* ← Length of *sorted_points*/2
3: **return** *sorted_points*[*mid*]

Time Complexity: $O(n \log n)$ (due to sorting)
Space Complexity: $O(n)$

1.3.2 The Mean Value of Our Data Points Across the Chosen Dimension

The mean value is used as the split position:

$$\mu_d = \frac{1}{n} \sum_{i=1}^{n} x_{i,d}$$

Algorithm 6 Mean Split

Require: *datapoints*, *dim*
Ensure: Mean value along the chosen dimension
1: $\mu \leftarrow$ Mean of points along dimension *dim*
2: **return** μ

Time Complexity: $O(n)$
Space Complexity: $O(1)$

1.3.3 Random Value in the Interval Between the Maximum and Minimum Values of Our Data Points Across the Chosen Dimension (Baseline)

A random value between the min and max values is chosen:

$$s_d = \text{rand}(\min(x_1, x_2, \ldots, x_n), \max(x_1, x_2, \ldots, x_n))$$

Algorithm 7 Random Split

Require: *datapoints*, *dim*
Ensure: Random value between min and max along the chosen dimension
1: *sorted_points* $\leftarrow$ Sort points along dimension *dim*
2: *split* $\leftarrow$ Random value in range $[sorted\_points[0], sorted\_points[-1]]$
3: **return** *split*

Time Complexity: $O(n)$ (to find min and max)
Space Complexity: $O(1)$

1.3.4 Geometric Center: (Maximum Value + Minimum Value) / 2

The midpoint between the min and max values is used:

$$\text{split position} = \frac{\min(x_1, x_2, \ldots, x_n) + \max(x_1, x_2, \ldots, x_n)}{2}$$

Algorithm 8 Geometric Center Split

Require: *datapoints*, *dim*
Ensure: Midpoint between min and max along the chosen dimension
1: *sorted_points* ← Sort points along dimension *dim*
2: *center* ← (*sorted_points*[0] + *sorted_points*[−1])/2
3: **return** *center*

Time Complexity: $O(n)$ (to find min and max)
Space Complexity: $O(1)$

1.4 Recursive Build of KD-Tree

The recursive build function constructs the KD-Tree by splitting the data at each node based on the chosen discriminant and split position.

Algorithm 9 Recursive Build of KD-Tree

Require: *datapoints*, *depth*, *last_dim*, $leaf\_size$, *max_depth*
Ensure: Subtree structure
1: **if** len(*datapoints*) ≤ $leaf\_size$ or (*max_depth* is not None and *depth* ≥ *max_depth*) **then**
2: **return** {"points": *datapoints*, "leaf": True, "depth": *depth*}
3: **end if**
4: Choose discriminant dimension using dimension choice algorithm
5: Choose split position using split position choice algorithm
6: Split *datapoints* into $left\_points$ and $right\_points$ based on the split position
7: $left\_child$ ← Recursive Build of KD-Tree($left\_points$, *depth* + 1, *split_dim*, $leaf\_size$, *max_depth*)
8: $right\_child$ ← Recursive Build of KD-Tree($right\_points$, *depth* + 1, *split_dim*, $leaf\_size$, *max_depth*)
9: **return** {"split_dim": *split_dim*, "split_val": $left\_child$, "right": $right\_child$, "depth": *depth*, "leaf": False}

Time Complexity: $O(n \log n)$ (recursive building)
Space Complexity: $O(n)$ (tree structure)

1.5 Complexity Summary for KD-Tree

Algorithm	Time Complexity	Space Complexity
Alternate Dimension	$O(1)$	$O(1)$
Random Dimension	$O(1)$	$O(1)$
Max Variance Dimension	$O(n \cdot k)$	$O(1)$
Widest Interval Dimension	$O(n \cdot k)$	$O(1)$
Median Split	$O(n \log n)$	$O(n)$
Mean Split	$O(n)$	$O(1)$
Random Split	$O(n)$	$O(1)$
Geometric Center Split	$O(n)$	$O(1)$
Recursive Build	$O(n \log n)$	$O(n)$

Table 1: Complexity of Discriminant Choice and Split Position Algorithms for KD-Tree

2 R-Tree

2.1 R-Tree Overview

An **R-Tree** is a balanced tree structure that partitions data into **hyperrectangles** (bounding boxes). Each node represents a hyperrectangle defined by **Pmin** (the lowest values for all dimensions) and **Pmax** (the highest values for all dimensions). Nearby points are grouped together in these hyperrectangles, which are recursively split as the tree grows.

2.2 Dividing the Data into Two Groups

The first step in creating an R-Tree involves dividing the data into two groups by selecting two **seeds**.

2.2.1 Choosing a Dimension and Selecting Two Farthest Points (Similar to KD-Tree's Discriminant)

A dimension is selected (using strategies like alternating, random, maximum variance, or widest interval), and the two farthest points along that dimension are chosen as seeds.

Algorithm 10 One Dimension Farthest Seeds

Require: *datapoints*, *nbr_dims*, *dimension_choice_alg*, *last_dim*
Ensure: Seeds that are farthest apart along one dimension
1: $dim \leftarrow$ call dimension choice algorithm
2: $max\_point \leftarrow$ point with max value in dimension dim
3: $min\_point \leftarrow$ point with min value in dimension dim
4: **return** $\{max\_point, min\_point\}$

Time Complexity: $O(n)$
Space Complexity: $O(1)$

2.2.2 Iterating Over All Pairs of Points and Choosing the Pair with the Longest Distance

The pair of points with the greatest Euclidean distance is selected as seeds.

Algorithm 11 Farthest Euclidean Distance Seeds

Require: *datapoints*, *nbr_dims*
Ensure: Seeds that are farthest apart based on Euclidean distance

```
max_dist ← 0
max_dist_points ← []
for each pair of points (point1, point2) do
    Compute Euclidean distance between point1 and point2
    if distance > max_dist then
        max_dist ← distance
        max_dist_points ← {point1, point2}
    end if
end for
return max_dist_points
```

Time Complexity: $O(n^2)$
Space Complexity: $O(1)$

2.3 Assigning Points to Groups

Once the seeds are chosen, the remaining points are assigned to one of the two groups.

2.3.1 Sacrifice Tree Balancing

Each point is assigned to the group whose seed is closest to it.

Algorithm 12 Closest Seed Group

Require: *seed*1, *seed*2, *datapoints*, *nbr_dims*
Ensure: Two groups based on proximity to seeds
1: *group*1 ← []
2: *group*2 ← []
3: **for** each point p in *datapoints* **do**
4: Compute distance from p to *seed*1 and *seed*2
5: **if** distance to *seed*1 < distance to *seed*2 **then**
6: Add p to *group*1
7: **else**
8: Add p to *group*2
9: **end if**
10: **end for**
11: **return** {*group*1, *group*2}

Time Complexity: $O(n)$
Space Complexity: $O(n)$

2.3.2 Sacrifice Clustering

Points are sorted by distance to one seed and divided into two equal-sized groups.

Algorithm 13 Sorting Distance to One Seed Group

Require: *seed*, *datapoints*, *nbr_dims*
Ensure: Two groups sorted by distance from the seed
1: *sorted_points* ← sort points by distance to *seed*
2: *group*1 ← first half of *sorted_points*
3: *group*2 ← second half of *sorted_points*
4: **return** {*group*1, *group*2}

Time Complexity: $O(n \log n)$ (due to sorting)
Space Complexity: $O(n)$

2.4 Tree Building algorithm

Algorithm 14 RTree.build

Require: *datapoints*
Ensure: Root of the RTree
1: Compute *Pmin* and *Pmax* for each dimension
2: Recursively build tree using seed and grouping choice algorithms
3: **return** root of the tree

Time Complexity: $O(n \log n)$ (recursive building)
Space Complexity: $O(n)$ (tree structure)

Algorithm 15 RTree.recursive.build

Require: *datapoints*, *depth*, *last_dim*
Ensure: Subtree structure
1: Select seeds using seed choice algorithm
2: Group points based on proximity to seeds
3: **for** each group **do**
4: $P_{min} \leftarrow$ the smallest coordinate values in each dimension (the "lower corner")
5: $P_{max} \leftarrow$ the largest coordinate values in each dimension (the "upper corner")
6: Recursively build subtrees for each group
7: **end for**
8: **return** subtree structure

Time Complexity: $O(n \log n)$ (recursive building)
Space Complexity: $O(n)$ (tree structure)

2.5 Complexity Summary for R-Tree

Algorithm	Time Complexity	Space Complexity
One Dimension Farthest Seeds	$O(n)$	$O(1)$
Farthest Euclidean Distance Seeds	$O(n^2)$	$O(1)$
Closest Seed Group	$O(n)$	$O(n)$
Sorting Distance to One Seed Group	$O(n \log n)$	$O(n)$
Recursive Build	$O(n \log n)$	$O(n)$

Table 2: Complexity of Seed Choice and Data Grouping Algorithms for R-Tree

2.6 Comparison of KD-Tree and R-Tree

Tree Type	Time Complexity	Space Complexity
KD-Tree	$O(n \log n)$	$O(n)$
R-Tree	$O(n^2)$	$O(n)$

Table 3: Comparison of Final Complexity for KD-Tree and R-Tree

3 Experiments on Tree-Based Indexing

In this section, we describe the general experimental process used to evaluate the performance of tree-based indexing structures, such as KD-Trees and R-Trees. The goal of these experiments is to analyze the **building time**, **clustering quality**, and **memory usage** of different tree variants. This process is applicable to both KD-Trees and R-Trees, and it provides a systematic way to compare the effectiveness of various configurations.

3.1 Experimental Process Overview

The experimental process consists of the following steps:

1. **Define Variants**:
   - We create different variants of the tree structure by varying key hyperparameters, such as:
     - **Dimension Choice**: The strategy for selecting the dimension to split on (e.g., random, max variance, widest interval, alternate).
     - **Split Position Choice**: The strategy for determining the split position along the chosen dimension (e.g., mean, median, random, geometric center).
     - **Leaf Size**: The maximum number of points allowed in a leaf node.
     - **Max Depth**: The maximum depth of the tree.
2. **Build the Tree**:
   - For each variant, we construct the tree using a subset of the dataset. The building process involves recursively splitting the data based on the chosen hyperparameters.
3. **Measure Building Time**:
   - We record the time taken to build the tree for each variant. This metric helps us understand the computational efficiency of different configurations.
4. **Evaluate Clustering Quality**:
   - We use the **Silhouette Score** to evaluate the quality of clustering produced by the tree. The Silhouette Score measures how well each point is clustered, considering both cohesion (how close points in the same cluster are) and separation (how far apart different clusters are). A higher score indicates better-defined clusters.
5. **Measure Memory Usage**:
   - We track the peak memory usage during the tree-building process. This metric helps us understand the memory efficiency of different variants.
6. **Analyze Results**:
   - We compare the results across variants to identify trends, trade-offs, and optimal configurations. This analysis helps us draw conclusions about the effectiveness of different hyperparameters.

3.2 Why Silhouette Score?

The **Silhouette Score** is chosen as the primary metric for evaluating clustering quality because of its several advantages:

- **Interpretability**: The score ranges from -1 to 1, where a higher value indicates better clustering. This makes it easy to interpret and compare results.
- **Cluster Cohesion and Separation**: The Silhouette Score considers both how close points are within the same cluster (cohesion) and how far apart different clusters are (separation). This dual consideration provides a comprehensive measure of clustering quality.
- **No Need for Ground Truth**: Unlike some other metrics, the Silhouette Score does not require ground truth labels, making it suitable for unsupervised learning tasks.
- **Scalability**: The Silhouette Score can be computed efficiently even for large datasets, making it practical for evaluating large tree structures.
- **Widely Accepted**: It is a standard metric used in many clustering evaluation tasks, allowing for easy comparison with other methods and studies.

3.3 General Observations

The experimental process reveals several key trade-offs that are common across tree-based indexing structures:

1. **Building Time vs. Clustering Quality**:
   - Variants that produce better clustering quality often require longer building times due to more complex splitting strategies (e.g., `max_variance` dimension choice).
   - Simpler strategies (e.g., `random` dimension choice) are faster to build but may result in poorer clustering quality.
2. **Memory Usage**:
   - Memory usage is generally influenced by the tree's depth and the number of points stored in each node. Variants with larger `leaf_size` or deeper trees may consume more memory.
3. **Hyperparameter Sensitivity**:
   - The choice of hyperparameters (e.g., dimension choice, split position, `leaf_size`, `max_depth`) has a significant impact on both the building time and clustering quality. Finding the right balance is crucial for optimizing performance.

3.4 Experiments on KD-Tree

In this section, we analyze the performance of different KD-Tree variants in terms of **building time**, **clustering quality (Silhouette Score)**, and **memory usage**. The goal is to understand how different choices of discriminant dimensions, split positions, `leaf_size`, and `max_depth` affect the efficiency and effectiveness of the KD-Tree.

3.4.1 Experimental Setup

We conducted experiments on a subset of the dataset (10,000 points) to evaluate the following variants of KD-Tree:

- **Dimension Choice**: `random`, `max_variance`, `widest_interval`, `alternate`
- **Split Position Choice**: `random`, `mean`, `median`, `geometric_center`
- **Leaf Size**: Ranged from 10 to 100
- **Max Depth**: Ranged from 5 to 45

For each variant, we measured:

1. **Building Time**: The time taken to construct the KD-Tree.
2. **Silhouette Score**: A metric that evaluates the quality of clustering, ranging from -1 (poor clustering) to 1 (well-defined clusters).
3. **Peak Memory Usage**: The maximum memory consumed during the construction of the KD-Tree.

3.4.2 Results and Analysis

Below is a detailed analysis of the results for each variant:

- **Variant 1**: `dimension_choice="random"`, `split_position_choice="random"`, `leaf_size=10`, `max_depth=None`
  - **Building Time**: 1.53 seconds
  - **Peak Memory**: 3.96 MB
  - **Silhouette Score**: -0.0838
  - **Analysis**: This variant is the fastest to build but has the lowest Silhouette Score, indicating poor clustering quality. The random choices for both dimension and split position likely lead to suboptimal tree structures.
- **Variant 2**: `dimension_choice="max_variance"`, `split_position_choice="mean"`, `leaf_size=20`, `max_depth=20`
  - **Building Time**: 533.77 seconds

  - **Peak Memory**: 2.49 MB
  - **Silhouette Score**: -0.0782
  - **Analysis**: This variant takes significantly longer to build due to the `max_variance` dimension choice, which requires computing variances for all dimensions. The Silhouette Score is slightly better than Variant 1 but still poor.

- **Variant 3**: `dimension_choice="widest_interval"`, `split_position_choice="median"`, `leaf_size=30`, `max_depth=15`
  - **Building Time**: 198.57 seconds
  - **Peak Memory**: 2.29 MB
  - **Silhouette Score**: -0.0754
  - **Analysis**: The `widest_interval` dimension choice reduces the building time compared to `max_variance`, but the clustering quality remains low. The `median` split position does not seem to improve the score significantly.

- **Variant 4**: `dimension_choice="alternate"`, `split_position_choice="geometric_center"`, `leaf_size=40`, `max_depth=10`
  - **Building Time**: 0.86 seconds
  - **Peak Memory**: 2.82 MB
  - **Silhouette Score**: 0.0879
  - **Analysis**: This variant performs well in terms of both building time and clustering quality. The `alternate` dimension choice and `geometric_center` split position seem to produce a more balanced tree structure, resulting in a positive Silhouette Score.

- **Variant 5**: `dimension_choice="random"`, `split_position_choice="mean"`, `leaf_size=50`, `max_depth=5`
  - **Building Time**: 0.26 seconds
  - **Peak Memory**: 1.53 MB
  - **Silhouette Score**: -0.0223
  - **Analysis**: This variant is very fast to build and uses minimal memory. The Silhouette Score is better than Variant 1, likely due to the `mean` split position, which provides a more balanced split compared to `random`.

- **Variant 6**: `dimension_choice="max_variance"`, `split_position_choice="median"`, `leaf_size=60`, `max_depth=25`
  - **Building Time**: 435.83 seconds
  - **Peak Memory**: 2.53 MB

  - **Silhouette Score**: -0.0743
  - **Analysis**: Similar to Variant 2, this variant suffers from high building time due to the `max_variance` dimension choice. The clustering quality remains poor, indicating that `max_variance` may not be the best choice for this dataset.

- **Variant 7**: `dimension_choice="widest_interval"`, `split_position_choice="geometric_center"` `leaf_size=70`, `max_depth=30`
  - **Building Time**: 524.57 seconds
  - **Peak Memory**: 3.67 MB
  - **Silhouette Score**: -0.0721
  - **Analysis**: This variant has a long building time and poor clustering quality, suggesting that the combination of `widest_interval` and `geometric_center` is not effective for this dataset.

- **Variant 8**: `dimension_choice="alternate"`, `split_position_choice="mean"`, `leaf_size=80`, `max_depth=35`
  - **Building Time**: 0.36 seconds
  - **Peak Memory**: 1.54 MB
  - **Silhouette Score**: -0.0335
  - **Analysis**: This variant is fast to build and uses minimal memory. The Silhouette Score is better than Variant 1, indicating that the `alternate` dimension choice and `mean` split position produce a more balanced tree.

- **Variant 9**: `dimension_choice="random"`, `split_position_choice="median"`, `leaf_size=90`, `max_depth=40`
  - **Building Time**: 0.36 seconds
  - **Peak Memory**: 1.55 MB
  - **Silhouette Score**: -0.0490
  - **Analysis**: Similar to Variant 5, this variant is fast to build but has a slightly lower Silhouette Score, likely due to the `random` dimension choice.

- **Variant 10**: `dimension_choice="max_variance"`, `split_position_choice="geometric_center"`, `leaf_size=100`, `max_depth=45`
  - **Building Time**: 2224.72 seconds
  - **Peak Memory**: 10.94 MB
  - **Silhouette Score**: -0.0198

– **Analysis**: This variant has the longest building time and highest memory usage, but the Silhouette Score is the best among all variants. This suggests that `max_variance` and `geometric_center` can produce better clustering quality, but at a significant computational cost.

3.4.3 Key Observations and Conclusions

1. **Building Time**:
   - Variants using `max_variance` or `widest_interval` for dimension choice take significantly longer to build due to the additional computations required.
   - Variants with `alternate` or `random` dimension choices are much faster to build.
2. **Clustering Quality**:
   - Variants with `alternate` dimension choice and `geometric_center` split position tend to have better Silhouette Scores.
   - Variants with `max_variance` dimension choice can produce better clustering quality but at a high computational cost.
3. **Memory Usage**:
   - Memory usage is generally low across all variants, with the exception of Variant 10, which uses significantly more memory due to the large `leaf_size` and `max_depth`.
4. **Trade-offs**:
   - There is a clear trade-off between building time and clustering quality. Variants that produce better clustering quality (e.g., Variant 10) take much longer to build.
   - Variants with `alternate` dimension choice and `geometric_center` split position (e.g., Variant 4) offer a good balance between building time and clustering quality.

3.4.4 Recommendations

- For applications where **building time** is critical, use variants with `alternate` or `random` dimension choices and `mean` or `geometric_center` split positions (e.g., Variant 4 or Variant 5).
- For applications where **clustering quality** is the primary concern, consider using `max_variance` dimension choice and `geometric_center` split position (e.g., Variant 10), but be prepared for longer building times and higher memory usage.