Towards Scalable Point-Cloud KNN Search: Efficient KD-Tree Construction via Sampled Median-of-Medians

Jiayi Xie

School of Automation, Nanjing University of Science and Technology, Nanjing, China

xiejy0920@163.com

Abstract. Autonomous driving applications in intelligent transportation, unmanned delivery and high-definition map navigation all require efficient spatial data processing for perception and localisation. Point cloud registration aligns scans obtained at different times or from different positions by repeatedly finding neighbouring points to establish correspondences. This operation is generally carried out by K-nearest neighbour (*KNN*) queries. KD-Tree is a general-purpose spatial index for *KNN* search. At the same time, each internal node selects a splitting axis and recursively divides its points at the median coordinate along that axis. The former way of sorting finds the median by arranging all the coordinates along the chosen axis. Multiple Sorts increase construction costs due to larger point clouds and deeper recursion. We propose MM-KDTree to reduce the cost. At each non-leaf node, we systematically sample 5% of the coordinates along the selected axis without replacement, compute the sample median using Median of Medians (MoM), and use it to partition the complete point set at that node. The recursive construction procedure and the exact *KNN* query rules are unchanged. Experiments show that MM-KDTree reduces construction time by 89.10% - 92.36% compared with a sorting-based KD-Tree and maintains the same exact top-10 results. Mean query latency is still in the same order of magnitude. Based on the above results, selecting split values from samples can reduce the cost of index construction while keeping the query accuracy.

Keywords: autonomous driving, *K*-nearest neighbor search, KD-Tree, Median of Medians.

1. Introduction

Road perception, vehicle localisation and high-definition map updates for autonomous driving all require accurate representations of the surrounding environment. Three-dimensional point clouds obtained by LiDAR are a typical type of these representations. Scans taken at different times or from different positions generally have different coordinate systems and need to be aligned by point cloud registration. The Iterative Closest Point (ICP) algorithm, for instance, iteratively finds correspondences and updates the spatial transformation [1] for registration. Correspondence search can generally be expressed as K-nearest neighbour (*KNN*) search: given a query point, *KNN* finds the K points closest to it in terms of Euclidean distance. Registration involves many such queries over multiple iterations, so *KNN* search efficiency directly affects the overall registration process.

A brute-force *KNN* search scans the complete point set, computing the distance from each query point to every point in the dataset. For a dataset of n points, a single query requires $O(n)$ distance evaluations. This cost is modest for small datasets, but repeated scans become expensive when a point cloud contains hundreds of thousands or millions of points. Practical systems therefore commonly build a spatial index before processing queries. The index records how points are organized in space, allowing a query to exclude regions that cannot contain a closer neighbor and continue searching the remaining regions. This reduces repeated scans of the complete point set. Spatial indices differ in how they organize space. R-Tree represents groups of objects by bounding regions, Ball-Tree organizes points hierarchically within nested hyperspheres, and iKD-Tree supports dynamic point insertion and deletion. Each has its own application settings. KD-Tree is a classical choice for exact nearest neighbor search on low-dimensional point data.

KD-Tree has a straightforward construction and query procedure. It organizes multidimensional points through recursive binary partitioning [2]. During construction, each internal node selects a coordinate axis and a split value, then assigns its points to two child nodes. A balanced KD-Tree typically uses the median coordinate along the selected axis as the split value. This produces left and

right subsets of similar size and helps maintain tree balance. A query first descends into the region containing the query point and backtracks only when the other side may contain a closer point, avoiding unnecessary distance evaluations.

Index construction costs must also be considered when improving query efficiency. Our sorting-based KD-Tree baseline sorts all coordinates along the selected axis at the current node and then takes the middle element. Yet partitioning requires only the median, not a complete ordering of the remaining coordinates. Sorting is also repeated at successive levels of recursive construction. After the root is partitioned, each child sorts its own coordinate sequence, and the same operation continues at subsequent levels. A point may therefore participate in sorting at several levels. Sorting a node containing s points takes $O(s \log s)$ time. Accumulating this cost across levels gives the sorting-based KD-Tree a construction complexity approaching $O(n \log^2 n)$ when the tree is approximately balanced. The cost of repeated sorting consequently increases with point-cloud size.

We propose MM-KDTree to address this construction cost. At each non-leaf node, our method samples a small fraction of the coordinates along the selected axis and computes their median using MoM. MoM selects an element without fully sorting the input sequence. It divides the sequence into small groups, finds each group median, and recursively selects a pivot from these medians. Selection then continues only in the subsequence containing the target rank [3]. MM-KDTree uses a 5% systematic sample and takes the sample median returned by MoM as the current split value. Sampling is confined to split-value selection: partitioning still processes all points at the node, and exact *KNN* query rules remain unchanged. Our main contributions are as follows:

1. At each non-leaf node of MM-KDTree, we systematically sample 5% of the coordinates along the selected axis. The sample size decreases with the number of points at the node.
2. We use MoM to select the sample median as the current split value, replacing the full coordinate sort used to obtain the median in a conventional sorting-based KD-Tree.
3. We evaluate spatial indices, sampling ratios, and construction parameters on five point-cloud benchmarks. MM-KDTree achieves a construction speedup of 9.17–13.08× over the sorting-based KD-Tree while preserving the same exact top-10 results.

1.1. Related Work

R-Tree organizes spatial objects using minimum bounding rectangles. Guttman's dynamic R-Tree [4] maintains the index through incremental insertion and node splitting. Beckmann et al. [5] subsequently used reinsertion and other strategies to reduce node overlap while jointly optimizing the area and perimeter of bounding rectangles. Other work has focused on construction strategies, such as STR bulk loading [6], whereas RLR-Tree [7] learns splitting strategies. Although these methods target different aspects of the index, R-Tree must still maintain bounding boxes and handle node overlap and splitting. During incremental construction, these costs accumulate as points are inserted.

KD-Tree recursively partitions multidimensional points into binary spatial regions and prunes queries using splitting planes or lower bounds on distances to regions. Friedman et al. [8] provided an early analysis of its nearest neighbor query performance. Most research has focused on query strategies and parallel implementations recently. Arya et al. [9] have studied approximate nearest neighbour search in fixed dimensions, and GGNN [10] and CAGRA [11] have explored GPU graph indices for faster nearest neighbour search. Zhou and others [12] have studied parallel KD-Tree construction on GPUs. Work on point clouds also includes nested spatial indices for LiDAR [13], parallel octree search [14], dedicated point-cloud *KNN* accelerators [15], and GPU-based search for point-cloud registration [16]. The above ways improve query organisation, storage structure or hardware execution efficiency. However, a static binary KD-Tree on a CPU still incurs repeated sorting costs if it determines median split values by fully sorting the coordinates at each level.

Median of Medians (MoM) constructs a pivot from group medians to select an element of a specified rank without fully sorting the input. Blum et al. [3] introduced the method and established its worst-case linear-time guarantee. The procedure divides the input into small groups, computes their medians, and recursively selects the median of those medians as a pivot. It then narrows the

search to the interval containing the target rank. For deterministic median selection, Dor and Zwick [17] improved the worst-case upper bound on the number of comparisons, while Alexandrescu [18] studied implementations aimed at practical runtime efficiency. These studies primarily address selection in general unordered sequences. In KD-Tree construction, applying standard MoM to the complete coordinate sequence at every internal node still repeats grouping, local sorting, and recursive selection as nodes are partitioned, accumulating costs across tree levels.

2. MM-KD Tree Method

Method overview. Our MM-KDTree uses a sampled MoM splitting strategy to reduce the construction cost of static point-cloud indices. Partitioning requires a split position, but neither a complete ordering of the coordinates nor an exactly equal division at every node. We therefore select a small sample of coordinates along the splitting axis and use MoM to obtain its median as the split value. Only the sample is processed during selection; the subsequent partition still covers the complete point set. MM-KDTree construction consists of cyclic axis selection, systematic coordinate sampling, sample-median selection, and recursive point-set partitioning. Figure 1 summarizes the procedure.

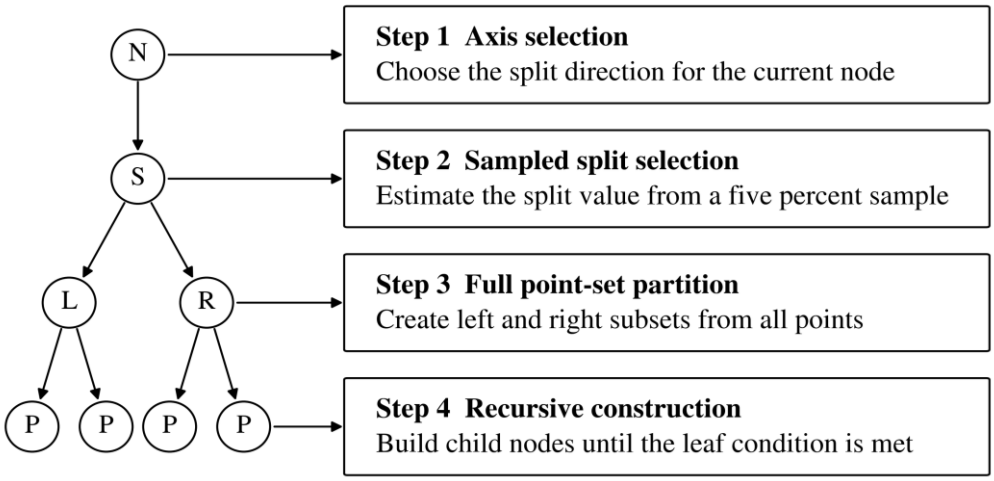


Figure 1. Overall construction procedure of MM-KDTree.

The sample median estimates the median split position of the complete coordinate sequence. Its deviation from the full median may affect subtree sizes and query costs. We examine this effect in the sampling-ratio experiments and use 5% as the default setting.

2.1. Split-Value Selection Using Sampled MoM

Coordinate sampling. Let P be the set of d-dimensional points at a node of depth h. We select the splitting axis as $a = h \bmod d$, cycling through the coordinate dimensions. The coordinates $p[a]$ of points p along axis a form the sequence X. Systematic sampling without replacement over the full index range of X produces the sample S. Its size is 5% of the sequence length, rounded up, as defined in (1).

$$X = \{p[a] \mid p \in P\}, S \subset X, \quad |S| = \lceil 0.05|X| \rceil \tag{1}$$

Systematic sampling places sample positions at regular intervals over the sequence index range and retrieves the corresponding coordinates without presorting. These intervals refer to sequence positions and do not imply uniform sampling in three-dimensional space. For a node containing s points, selection processes approximately $0.05s$ coordinates, so its input size decreases with the node size.

Grouping and pivot selection. We select the sample median at target rank $k = \lfloor |S|/2 \rfloor$, using zero-based ranks. MOM_SELECT(A, k, g) is the selection subroutine called by the construction procedure. Here, A denotes the current selection sequence, with $A = S$ on the initial call. When $|A| > g$, we divide

A into t groups G_j, each containing at most g elements. Their medians form the sequence M. We recursively select the median of M as the pivot q for the current selection step, as shown in (2). All group medians, including duplicates, are retained in M.

$$M = \{\text{median}(G_j) \mid j = 1, \dots, t\}, q = \text{MOM_SELECT}(M, \lfloor |M|/2 \rfloor, g). \quad (2)$$

Selection by target rank. The pivot q partitions A into A_1, A_2, and A_3, containing elements smaller than, equal to, and greater than q, respectively. If $k < |A_1|$, selection continues in A_1 with the same target rank. If $|A_1| \le k < |A_1| + |A_2|$, the subroutine returns q. Otherwise, selection continues in A_3 with the target rank updated to k' according to (3). Each step processes only the subsequence that still contains the target element.

$$A_1 = \{x \in A | x < q\}, A_2 = \{x \in A | x = q\}, A_3 = \{x \in A | x > q\}, k' = k - |A_1| - |A_2| \quad (3)$$

Determining the split value. If the current sequence has g or fewer elements, sort this small sequence by the subroutine and return the element at the target rank. After selection, it returns the median of S to the construction procedure as the node's split value q. q is a local pivot in recursive selection, and the value returned by the top-level call is the required split value. For an even-length sequence, we take the maximum of the two middle ranks.

2.2. Point-Set Partitioning and Recursive Construction

Point-set partitioning. Given the splitting axis a and split value q, we partition the complete point set P according to the relation between $p[a]$ and q. Coordinates on either side need no further sorting. Equation (4) gives the initial assignment: points equal to q are placed on the right and then reassigned according to the rule below.

$$P_l = \{p \in P \mid p[a] < q\}, P_r = \{p \in P \mid p[a] \ge q\}. \quad (4)$$

Handling equal coordinates. For points with p[a] = q, change their assignment so that the two subsets have nearly the same size, and keep the points strictly below and strictly above q on their respective sides. If the desired equal-size cut falls within the block of equal coordinates, then split at that position. Otherwise, we use the boundary of that block closest to the desired cut. This avoids assigning all equal-coordinate points to one side and thus produces an empty subtree. Each point belongs to exactly one subset, and the two subsets together cover P.

Recursion and termination. Partitioning is used to build the left and right subtrees, and then the depth is increased by one. At the beginning of each construction call, if the number of points does not exceed the leaf capacity L, a leaf is created to store the subset. Otherwise, the split value is calculated as in Section 2.1 and partitioning continues. Each branch terminates upon satisfying the leaf capacity condition. Since the sampled splits do not guarantee equal-sized subsets, the height of the resulting tree will be different depending on where the splits occur.

Algorithm: BUILD_MM_KDTREE	
Input: point set P; depth h; leaf capacity L; group size g Output: root node v	
1	if $\|P\| \leq L$ then
2	↳ return LEAF(P);
3	$a \leftarrow h \bmod d$; $X \leftarrow \{p[a] \mid p \in P\}$;
4	$S \leftarrow$ SYSTEMATIC_SAMPLE(X, 5%);
5	$q \leftarrow$ MOM_SELECT(S, $\lfloor \|S\|/2 \rfloor$, g);
6	(P_left, P_right) $\leftarrow$ PARTITION(P, a, q);
7	$v \leftarrow$ NODE(a, q);
8	v.left $\leftarrow$ BUILD_MM_KDTREE(P_left, h+1, L, g);
9	v.right $\leftarrow$ BUILD_MM_KDTREE(P_right, h+1, L, g);
10	return v;
Function: MOM_SELECT(A, k, g)	
11	if $\|A\| \leq g$ then
12	↳ return SORT(A)[k];
13	Partition A into groups G_j, each containing at most g elements;
14	for each group G_j do
15	↳ $m_j \leftarrow$ MEDIAN(G_j);
16	$M \leftarrow \{m_j\}$; $q \leftarrow$ MOM_SELECT(M, $\lfloor \|M\|/2 \rfloor$, g);
17	(A_1, A_2, A_3) $\leftarrow$ PARTITION(A, q);
18	if $k < \|A_1\|$ then
19	↳ return MOM_SELECT(A_1 , k, g);
20	else if $k < \|A_1\| + \|A_2\|$ then
21	↳ return q;
22	**else**
23	↳ return MOM_SELECT(A_3 , k-$\|A_1\|$-$\|A_2\|$, g);

Algorithm description. The algorithm shows the construction process of our proposed MM-KDTree. BUILD_MM_KDTREE takes the point set P, depth h, leaf capacity L, and group size g as input, and returns the subtree root v. The subroutine MOM_SELECT(A, k, g) returns the element of A at zero-based index k as the split value. The statements below follow the call sequence: the main procedure calls selection, and then resumes tree construction after the subroutine returns.

Lines 1–3. The main procedure checks the leaf condition. It returns a leaf if $|P| \leq L$; otherwise, it selects axis a and extracts the coordinate sequence X.

Lines 4–5. Systematic sampling selects 5% of X to form S. The procedure calls MOM_SELECT with $A = S$ and $k = \lfloor |S|/2 \rfloor$, entering the selection subroutine in lines 11–23.

Lines 11–12. The subroutine checks the sequence length. If $|A| \leq g$, it sorts A and returns the element at rank k.

Lines 13–16. A longer sequence is divided into groups. Their medians m_j form M, and a recursive MOM_SELECT call selects the current pivot q.

Line 17. The pivot q partitions the current sequence into A_1, A_2, and A_3, containing elements smaller than, equal to, and greater than q, respectively.

Lines 18–21. If the target rank falls in A_1, selection continues there. If it falls in A_2, the subroutine returns q.

Lines 22–23. If the target rank falls in A_3, selection continues there after subtracting the lengths of A_1 and A_2 from the rank. When the top-level subroutine call returns, line 5 assigns the sample median to the split value q.

Lines 6–7. The main procedure uses a and q to partition the complete point set P into left and right subsets. It creates the internal node v, recording the axis and split value.

Lines 8–10. The main procedure recursively builds both subtrees, assigns their roots to v.left and v.right, and returns v.

3. Experimental Setup

3.1. Baselines

We compare MM-KDTree with a sorting-based KD-Tree, R-Tree, Ball-Tree, and iKD-Tree. The sorting-based KD-Tree fully sorts the selected-axis coordinates at every internal node. R-Tree uses the three-dimensional incremental implementation in libspatialindex with Guttman's quadratic split. Our Ball-Tree implementation uses recursive partitioning with two pivots; early construction methods for ball hierarchies were summarized by Omohundro [19]. For iKD-Tree [20], we use the official hku-mars implementation. MM-KDTree and the sorting-based KD-Tree share the same point sets, axis-selection rule, leaf capacity, and exact query routine. They differ only in how split values are computed.

3.2. Datasets

The source point clouds come from two public autonomous driving datasets. Road scenes use Velodyne frames from the KITTI 3D object detection benchmark [21], and urban scenes use LIDAR_TOP scans from nuScenes-mini [22]. Only the three-dimensional coordinates (x, y, z) are retained and loaded in float32 format.

We select non-overlapping files from these datasets to form five benchmarks: OpenRoad, RoadCore, UrbanFlow, UrbanCore, and UrbanMega. The first two use KITTI training/velodyne frames, and the remaining three use nuScenes-mini LIDAR_TOP scans. UrbanMega comprises the remaining 323 frames, totaling 11,214,208 points. Frames are concatenated directly in their sensor coordinate frames without global multi-frame registration. For each benchmark, 100 query points are selected without replacement using a fixed random seed. Table 1 lists the scene types and point counts.

Table 1. Scene types and point counts of the five benchmarks.

Benchmark	Scene type	Number of points
OpenRoad	Open road	362,543
RoadCore	Central road area	709,614
UrbanFlow	Urban area	1,111,040
UrbanCore	Urban core	1,700,960
UrbanMega	Aggregated urban scans	11,214,208

3.3. Evaluation Metrics

Index efficiency is measured by construction time and mean query time; that is, the one-time cost to build an index and the latency of queries. Query correctness is verified for K = 10. For each of the 100 fixed query points, we compare the ten Euclidean distances returned by the index with those obtained from a blockwise exhaustive search over the complete point set. Results are classified as exact top-10 when the distances agree within relative and absolute tolerances of 2×10^{-5}. This criterion does not require the same neighbour identifiers and is not a recall metric for approximate search. Each index is built three times independently. After each build, one query is run as a warm-up, followed by five timed batches of 100 queries. Divide the median batch time by 100 to get the mean latency per query. The first timed batch after each build is checked separately against brute

force, and this includes iKD-Tree. Figures show the mean and sample standard deviation of the three independent builds.

3.4. Experimental Environment and Default Parameters

All experiments were rerun in the same environment: an AMD EPYC 9V74 processor, nine logical CPUs available to the process, a container CPU quota equivalent to eight cores, a 20 GiB memory limit, and Linux kernel 6.18.35. The Python 3.12.13 implementations use NumPy 2.3.5, Numba 0.67.0, and Rtree 1.4.1/libspatialindex. The official iKD-Tree implementation was compiled with g++ 13.3.0 using -O3 and C++17. Timing excludes data loading, array copying, JIT warm-up, query-point selection, and brute-force verification. All methods use identical point sets, query points, Euclidean distances, and $K = 10$. Configurations were executed sequentially.

KD-Tree, Ball-Tree, and MM-KDTree use a leaf capacity of 30. The MoM group size g is 5, and the sampling ratio is 5%. A selection sequence of length at most g is sorted directly to return the target-rank element. MM-KDTree stores only the splitting axis, split value, and child links at internal nodes; queries use splitting-plane pruning. R-Tree uses a capacity of 30 for both leaf and internal nodes and a minimum fill ratio of 0.4. iKD-Tree retains the official single-point node structure and default balancing logic. These settings remain fixed except in the parameter experiments.

4. Construction Efficiency

We compare construction times for the five spatial indices on all five benchmarks. Each index is built independently three times, and the error bars in Figure 2 represent one standard deviation. The vertical axis is logarithmic because incremental R-Tree construction is much slower than the other methods. At each non-leaf node, MM-KDTree applies MoM to only 5% of the selected-axis coordinates, then scans the complete point set to form two partitions. This avoids repeated full sorting of the selected-axis coordinates at internal nodes.

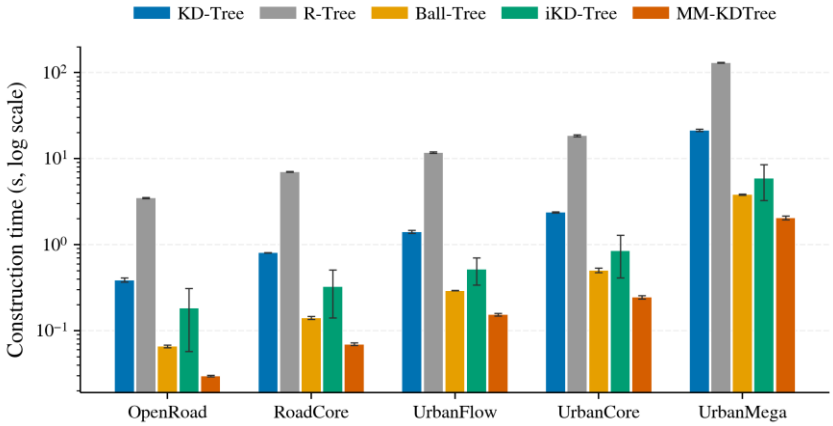


Figure 2. Index construction time.

Analysis. Figure 2 shows that construction times generally increased with point-cloud size, while the relative ordering of the five methods remained consistent. MM-KDTree had the shortest construction time on every benchmark, achieving a speedup of 9.17–13.08× over the sorting-based KD-Tree and reducing construction time by 89.10%–92.36%. On UrbanMega, the largest benchmark, mean construction time was 2.04 s for MM-KDTree and 21.37 s for the sorting-based KD-Tree. The construction advantage therefore persisted at a scale of more than ten million points. Ball-Tree, iKD-Tree, and R-Tree also took longer to build than MM-KDTree, with R-Tree requiring the most time.

5. Query Performance

On each benchmark, all five indices perform exact *KNN* search with $K = 10$ for the same 100 query points. Each independently built index processes five query batches. We take the median batch time

and convert it to mean time per query. Figure 3 reports the mean across three builds, with error bars indicating one standard deviation. Query timing includes tree traversal and candidate-distance processing, but excludes index construction and brute-force verification.

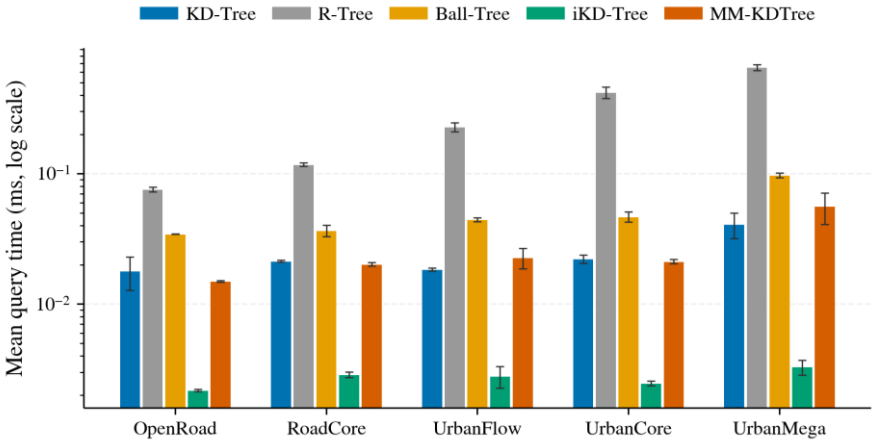


Figure 3. Mean query time.

Analysis. In all verified batches, the top-10 distances returned by all five indices agreed with brute force within the specified tolerances. MM-KDTree had mean query times of 0.0149–0.0560 ms, within the same order of magnitude as the sorting-based KD-Tree. Its mean query time was no higher than that of the sorting-based KD-Tree on three benchmarks, but was 23.12% higher on UrbanFlow and 36.99% higher on UrbanMega. These differences in means were not tested for statistical significance and do not establish a consistent query advantage. Together with the construction results, they show that the main benefit of MM-KDTree lies in static index construction.

6. Sampling Ratio and Parameter Analysis

6.1. Sampling Ratio

We evaluate systematic sampling without replacement at ratios of 5%, 10%, 20%, 30%, 40%, and 50% on all five benchmarks. All settings use the same leaf capacity, MoM group size, query points, three-build protocol, and correctness checks. Because the sampling ratio applies only to MM-KDTree, Figure 4 reports only its results. The fixed iKD-Tree results are shown in Figures 2 and 3.

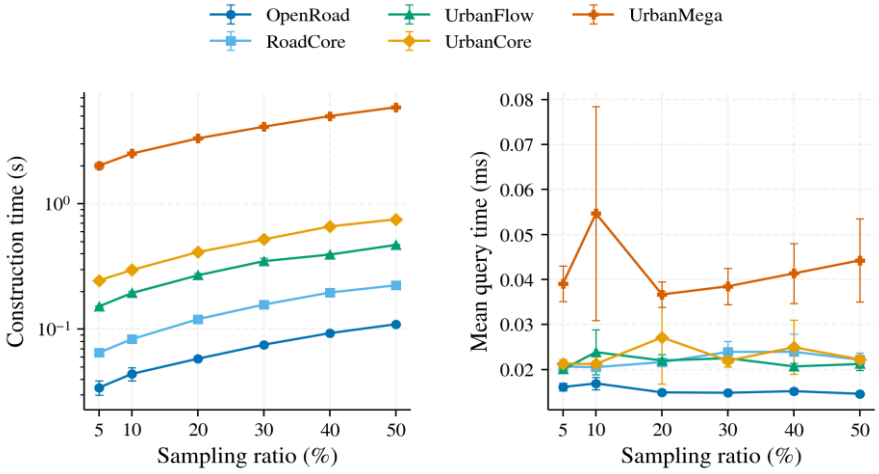


Figure 4. Effect of sampling ratio on construction and exact query times.

Analysis. A sampling ratio of 5% produced the lowest mean construction time on every benchmark. At 50%, construction took 2.92–3.44 times as long as at 5%. Across the tested ratios, the range of mean query times was 0.0023–0.0180 ms, depending on the benchmark. Query time did not consistently improve as the sampling ratio increased. Considering both construction cost and query

performance, we use 5% as the default. This is an empirical choice within the tested range, not a theoretical optimum for all point distributions.

6.2. Leaf Capacity and MoM Group Size

With the sampling ratio fixed at 5%, we vary the leaf capacity L over 10, 20, 30, 50, and 100, and the MoM group size g over 3, 5, 7, 9, and 11. Figure 5 reports construction and exact query times on the five benchmarks. The parameter L affects tree height and the number of points stored in each leaf, while g controls local grouping and recursion within MoM. Figure 5 contains only the MM-KDTree parameter results; the fixed iKD-Tree results appear in Figures 2 and 3.

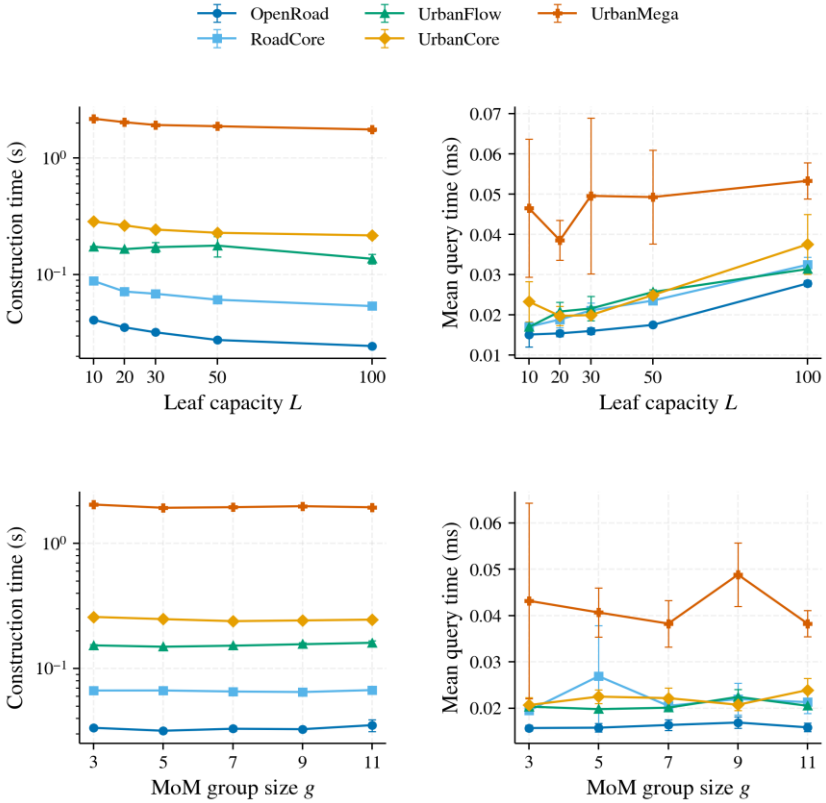


Figure 5. Effects of leaf capacity and MoM group size on MM-KDTree.

Analysis. Increasing L from 10 to 100 reduced construction time by 19.3%–40.5% and increased mean query time by 14.6%–90.2%. Larger leaves require fewer internal-node splits but more distance evaluations during leaf-level queries. We use $L = 30$ to balance these costs; it is not uniformly optimal across benchmarks. For a fixed point set and fixed sample positions, different group sizes g return the same sample median, so their main effect is on the computational cost of selection. Figure 5 reports the measured differences. We retain $g = 5$ as the default without claiming that it is optimal across all configurations.

7. Conclusion

We proposed MM-KDTree to reduce the construction cost of a sorting-based KD-Tree through sampled MoM split selection. Each non-leaf node chooses its splitting axis according to depth, systematically samples 5% of the corresponding coordinates without replacement, and calls MOM_SELECT to obtain the sample median. This split value partitions the complete point set, after which both subsets are built recursively. Queries retain the exact backtracking rules. The complete experimental rerun showed a construction-time reduction of 89.10%–92.36% relative to the sorting-based KD-Tree, corresponding to a speedup of 9.17–13.08×. In every verified batch, the top-10 distances agreed with brute force within the specified tolerances. Mean query times for MM-KDTree were 0.0149–0.0560 ms, within the same order of magnitude as those of the sorting-based KD-Tree.

References

[1] P. J. Besl and N. D. McKay, “A Method for Registration of 3-D Shapes,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 14, no. 2, pp. 239–256, 1992.

[2] J. L. Bentley, “Multidimensional Binary Search Trees Used for Associative Searching,” Communications of the ACM, vol. 18, no. 9, pp. 509–517, 1975.

[3] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, “Time Bounds for Selection,” Journal of Computer and System Sciences, vol. 7, no. 4, pp. 448–461, 1973.

[4] A. Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching,” in Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, pp. 47–57, 1984.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, “The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles,” in Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, pp. 322–331, 1990.

[6] S. T. Leutenegger, M. A. Lopez, and J. Edgington, “STR: A Simple and Efficient Algorithm for R-Tree Packing,” in Proceedings of the 13th International Conference on Data Engineering, pp. 497–506, 1997.

[7] T. Gu, K. Feng, G. Cong, C. Long, Z. Wang, and S. Wang, “The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data,” Proceedings of the ACM on Management of Data, vol. 1, no. 1, Article 63, pp. 1–26, 2023.

[8] J. H. Friedman, J. L. Bentley, and R. A. Finkel, “An Algorithm for Finding Best Matches in Logarithmic Expected Time,” ACM Transactions on Mathematical Software, vol. 3, no. 3, pp. 209–226, 1977.

[9] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, “An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions,” Journal of the ACM, vol. 45, no. 6, pp. 891–923, 1998.

[10] F. Groh, L. Ruppert, P. Wieschollek, and H. P. A. Lensch, “GGNN: Graph-Based GPU Nearest Neighbor Search,” IEEE Transactions on Big Data, vol. 9, no. 1, pp. 267–279, 2023, doi: 10.1109/TBDATA.2022.3161156.

[11] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, and Y. Wang, “CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs,” in 2024 IEEE 40th International Conference on Data Engineering (ICDE), pp. 4236–4247, 2024, doi: 10.1109/ICDE60146.2024.00323.

[12] K. Zhou, Q. Hou, R. Wang, and B. Guo, “Real-Time KD-Tree Construction on Graphics Hardware,” ACM Transactions on Graphics, vol. 27, no. 5, Article 126, 2008.

[13] C. J. Ogayar-Anguita, A. López-Ruiz, A. J. Rueda-Ruiz, and R. J. Segura-Sánchez, “Nested Spatial Data Structures for Optimal Indexing of LiDAR Data,” ISPRS Journal of Photogrammetry and Remote Sensing, vol. 195, pp. 287–297, 2023, doi: 10.1016/j.isprsjprs.2022.11.018.

[14] F. Chen, R. Ying, J. Xue, F. Wen, and P. Liu, “ParallelNN: A Parallel Octree-Based Nearest Neighbor Search Accelerator for 3D Point Clouds,” in 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 403–414, 2023, doi: 10.1109/HPCA56546.2023.10070940.

[15] M. Han, L. Wang, L. Xiao, H. Zhang, T. Cai, J. Xu, Y. Wu, C. Zhang, and X. Xu, “BitNN: A Bit-Serial Accelerator for K-Nearest Neighbor Search in Point Clouds,” in Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA), pp. 1278–1292, 2024.

[16] Q. Chang, W. Wang, and J. Miyazaki, “Accelerating Nearest Neighbor Search in 3D Point Cloud Registration on GPUs,” ACM Transactions on Architecture and Code Optimization, vol. 22, no. 1, Article 43, pp. 1–24, 2025, doi: 10.1145/3716875.

[17] D. Dor and U. Zwick, “Selecting the Median,” SIAM Journal on Computing, vol. 28, no. 5, pp. 1722–1758, 1999.

[18] A. Alexandrescu, “Fast Deterministic Selection,” in Proceedings of the 16th International Symposium on Experimental Algorithms (SEA), Leibniz International Proceedings in Informatics, vol. 75, Article 24, pp. 24:1–24:19, 2017.

[19] S. M. Omohundro, Five Balltree Construction Algorithms, Technical Report TR-89-063, International Computer Science Institute, 1989.

[20] Y. Cai, W. Xu, and F. Zhang, “ikd-Tree: An Incremental K-D Tree for Robotic Applications,” arXiv:2102.10808, 2021.

[21] A. Geiger, P. Lenz, and R. Urtasun, “Are We Ready for Autonomous Driving? The KITTI Vision Benchmark Suite,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 3354–3361, 2012.

[22] H. Caesar et al., “nuScenes: A Multimodal Dataset for Autonomous Driving,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 11618–11628, 2020.