**Journal of Computing and Electronic Information Management**
ISSN: 2413-1660 | Vol. 12, No. 2, 2024

# An algorithm for constructing spatial vector data storage based on KD-tree and density estimation

**Ning Wang**

School of Computer Science and Technology, Qingdao University, Qingdao 266071, China
349586886@qq.com

**Abstract:** With the widespread application of spatial vector data in various fields, this paper proposes a novel algorithm for constructing spatial vector data storage. The algorithm is based on KD-tree and density estimation techniques, aiming to improve the storage efficiency and query performance of large-scale spatial vector data. The algorithm first efficiently organizes spatial vector data using KD-tree, and then performs density estimation based on the Locality Sensitive Hashing (LSH) algorithm. Algorithm optimizations for spatial partitioning are applied to the KD-tree construction algorithm, reducing the search range during queries and improving retrieval speed. By testing with Green Tide data, the algorithm based on KD-tree and density estimation demonstrates higher query efficiency and better scalability across multiple test datasets. Particularly, when dealing with large-scale datasets characterized by non-uniform data distributions, this algorithm significantly improves data retrieval speed while maintaining low storage overhead.

**Keywords:** Spatial vector data; KD-tree; Density estimation; Locality-sensitive hashing.

## 1. Introduction

In the era of data-driven advancements, spatial vector data plays a crucial role in various fields, including Geographic Information Systems (GIS), Computer-Aided Design (CAD), and Virtual Reality (VR), among others. These applications place high demands on data storage and retrieval efficiency, especially when dealing with large-scale datasets. Traditional spatial data storage methods face challenges regarding rapid data growth and inefficient query performance [1].

To address these issues, we introduce the KD-tree as the foundation for our data storage structure. The KD-tree, being a classical spatial data structure, organizes data points into a tree-like structure by partitioning the space. This enables accelerated range queries and nearest neighbor searches. This data structure is capable of better accommodating the characteristics of spatial vector data, thereby improving query efficiency. However, individual KD-trees still have room for improvement in efficiency when dealing with uneven data distributions. To address this issue, this paper introduces a density estimation based on locality sensitive hashing algorithm to optimize the spatial partitioning of the KD-tree construction algorithm.

In spatial vector data, there exist clusters of data with varying densities. By estimating the density of the data, we can partition the data more accurately, effectively distinguishing high-density areas from low-density areas. This allows for flexible allocation of resources during storage, leading to improved query efficiency and data retrieval speed.By conducting experiments on actual green tide data, it has been demonstrated that the proposed algorithm exhibits significant advantages over traditional methods in terms of storage efficiency and query speed. This provides strong support for further advancements in the field of vector spatial data management.

## 2. Data Storage Algorithm for KD-Tree Spatial Index Structure.

The KD-tree [2] partitions the space by dividing it based on the dimensional value of data points at each node, making it highly efficient for range and nearest neighbor searches. It exhibits outstanding performance when handling point set data, such as point cloud data in computer graphics. By alternately selecting a splitting point on different dimensions, the data is divided into two parts, allowing for the recursive construction of the tree. Each node represents a data point in the corresponding dimension, where the left subtree contains points with smaller values in the selected dimension, and the right subtree contains points with larger values. The construction of a KD tree follows a divide-and-conquer strategy, where the optimal splitting dimension is selected to evenly partition the dataset into subspaces in the space, thus better accommodating the distribution of dense data.

Fig.1 depicts the construction process of the data storage structure of a KD tree. The steps can be broadly divided into the following:

1.Determining the splitting axis and node for the current data:

Given a data sample set S, the initial splitting axis is set to r=0. If the number of data points in S is greater than 1, it is common practice to sort all the points in S based on the size of the rth coordinate. The selected median element from this arrangement is then chosen as the feature coordinate for the current node, and the splitting axis r is recorded.

2.Building the left and right subtrees of a KD-tree structure:

Let SL be the set of elements in S that are arranged before the median element. Let SR be the set of elements in S that are arranged after the median element. The left branch of the current node is constructed as a kd-tree with SL as the dataset and r as the splitting axis. The right branch of the current node is constructed as a kd-tree with SR as the dataset and r as the splitting axis.

3.Iteratively construct and output the root node.

Split the dataset iteratively along dimensions using the modulo operation mod n, where n represents the total number of dimensions. Set r ← (r+1) mod n to cyclically partition the dataset until the final dataset count reaches 1. Output the root node to complete the construction of the KD tree.

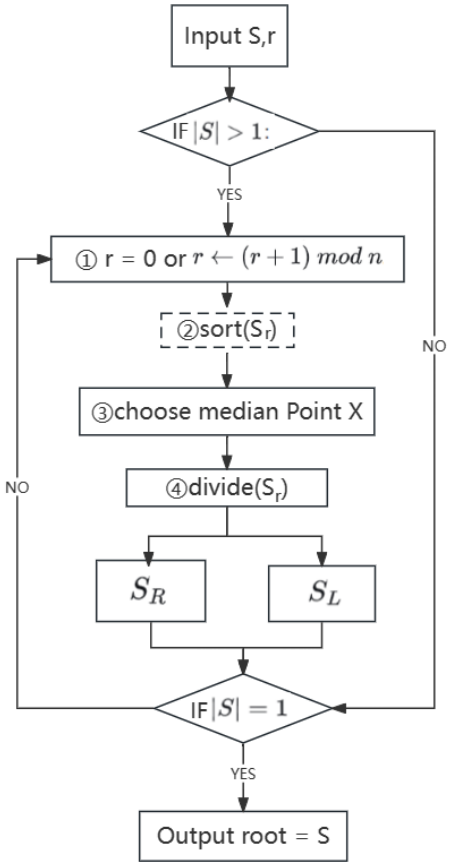


**Fig.1** KD Tree Construction Process

# 3. Optimizing the construction algorithm for density estimation implementation based on Locality Sensitive Hashing algorithm

The key to guaranteeing the performance of the KD-tree construction algorithm lies in the selection of the splitting axis. Traditional KD-trees typically use the median for spatial partitioning, which works well in the case of uniformly distributed data. However, in the presence of uneven data distribution, the median fails to achieve reasonable spatial partitioning, resulting in an imbalanced distribution of the constructed tree structure and affecting subsequent retrieval and loading speeds [3].

In order to optimize and improve this issue, this paper proposes a method based on data density to determine the splitting positions, as well as the splitting axis and points. This approach ensures that the resulting subtrees contain roughly the same number of points, thus maintaining the balance of the tree and effectively handling uneven data distribution. Consequently, it enhances the loading performance of retrieving unevenly distributed data.

The algorithm for constructing KD-trees based on data density partitioning primarily determines how to divide the data space by considering the number of neighboring points around each data point. This process aims to create a KD-tree. When calculating data density, both accuracy and computational efficiency need to be considered. Therefore, this paper utilizes the Locality Sensitive Hashing (LSH) algorithm for density estimation. The density estimation values are used as a basis for subsequent data sorting.

## 3.1. Basic Theory of Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) is an algorithm used for efficient approximate similarity search. The use of LSH for computing data point densities is particularly suitable for large-scale datasets as it avoids the computation of actual distances between all pairs of data points. LSH achieves this by mapping data points to a hash space, effectively reducing the complexity of processing the data and providing fast query responses in big data environments. The core idea of LSH is to map data points to a lower-dimensional hash space, such that similar data points have a higher probability of being mapped to the same "bucket" or hash value. When computing the density of spatial vector data points, LSH can quickly identify approximate neighbors for each data point. By counting the number of neighbors within its hash bucket, the density of a point can be approximated.

## 3.2. Optimizing KD-tree construction algorithm based on density estimation using LSH

Here are the detailed steps for optimizing this algorithm:

Step 1: Generate MinHash signatures for each data point

Using the MinHash algorithm, generate a MinHash signature for each data point. The computation of MinHash signatures involves the following steps.

Selecting hash functions: Choose a set of hash functions h1, h2,..., hk, where k is the length of the MinHash signature. This number affects the accuracy of the MinHash signature and the memory consumption in the LSH process. In our experiments, we set k to be 128.

Computing MinHash Values: For each set S, calculate its MinHash signature M(S). The MinHash signature is a vector consisting of k hash values hi(S), where each hash value is obtained by applying the hash function hi to the data point S.

Specifically, for each hash function hi, the MinHash value mi(S) is defined as the minimum value obtained by applying the hash function to all elements in set S.

$$m_i(S) = \min\{h_i(x) \mid x \in S\} \tag{1}$$

This means that for each h_i, we find the minimum hash value obtained by applying h_i to all elements in set S. Finally, these MinHash values form the MinHash signature M(S) of set S.

$$M(S) = [m_1(S), m_2(S), \ldots, m_k(S)] \tag{2}$$

This signature is a series of hash values that represent the "fingerprint" of the original set.

Step 2: Constructing the LSH Model

Using the generated MinHash signatures, we construct the LSH model, which maps data points with similar MinHash signatures into the same "bucket" or hash value. The similarity determination criterion relies on estimating the Jaccard similarity and the specific process is as follows:

For example, given two sets S1 and S2, their Jaccard similarity can be estimated by comparing the proportion of equal values in their MinHash signatures. The specific formula is as follows:

$$J(S_1, S_2) \approx \frac{\left|\{i \mid m_i(S_1) = m_i(S_2)\}\right|}{k} \quad (3)$$

J(S1, S2) represents the Jaccard similarity between sets S1 and S2, while mi(S1) and mi(S2) correspond to the MinHash values of sets S1 and S2 respectively, based on hash function hi. In practical experiments, the threshold for defining the similarity between two data points is set at 0.9, meaning that they are considered similar only when their MinHash signatures have a similarity exceeding 90%.

Step 3: Querying Approximate Neighbors

For each data point, the LSH model is queried to identify other data points with similar MinHash signatures. The number of these similar data points can be considered as the approximate local density of that point. By calculating the number of approximate neighbors for each data point, a list reflecting the density of each green tide data point is obtained.

Step 4: Constructing KD-Tree based on Density List

Based on the obtained density list, the median of the sorted list is selected as the partition threshold to serve as the current node of the KD-tree. By iterating through this process, the final KD-tree spatial indexing structure is constructed.

In summary, the density estimation and KD-tree construction based on the LSH algorithm are illustrated in Algorithm 1.

Algorithm 1: Density Estimation and KD-Tree Construction based on Locality Sensitive Hashing (LSH) Algorithm Input: Data points D={d1, d2,..., dn}; LSH functions H={h1, h2, ..., hk}; KD-tree partition threshold T Output: KD-tree

1)Initialize an empty LSH table L.

2)For each data point di∈D, perform steps 3-5.

3)Initialize an empty MinHash signature Mi.

4)For each hash function hj∈H, perform step 5.

5)Update Mi as the minimum value on hj(di).

6)Insert (di,Mi) into LSH table L.

7)End loop.

8)Initialize an empty list S to store sorted data points.

9)For each data point di∈D, perform steps 10-12.

10)Query approximate neighbors Ni of di using LSH table L.

11)Calculate the density value ρi of di as the count of Ni.

12)Add (di,ρi) to S.

13)End loop.

14)Sort the elements in S based on the density value ρ.

15)Construct a KD-tree using the sorted data points S, with a partition threshold of the median T.

16)Return the constructed KD-tree

# 4. Experiment

The experiment is divided into two parts. The first part involves the implementation of the data storage construction algorithm based on KD-tree. The second part focuses on the optimization of the spatial partitioning of the KD-tree spatial indexing structure based on density estimation when dealing with unevenly distributed data. A comparison is made with the original KD-tree construction algorithm to validate the performance improvement of the optimized algorithm. Since there may be some variations in the results obtained from each experiment, the final experimental data is the average of the results obtained from 20 experiments.

## 4.1. Experimental Data

In this study, we selected the green tide data in the Yellow Sea region during the green tide outbreak period from May to August for each year from 2019 to 2023 as spatial vector data samples. Satellite images of the data can be freely downloaded from the China Ocean Satellite Data Service System (https://osdds.nsoas.org.cn/#/). After image processing using ENVI software, specific algorithms were applied to extract the green tide images from the satellite images. Finally, the latitude and longitude information of the green tide data points in the extracted images were collected and compiled into a daily-based coordinate data table, which was stored in JSON format

## 4.2. Performance Verification Experiment of KD-Tree Data Storage and Construction Algorithm

To verify the efficient performance of the KD-tree spatial indexing structure in storing spatial vector data, this study employed four different methods: direct loading, quadtree loading, R-tree loading, and KD-tree loading. These methods were used to construct data models, perform data traversal and extraction, and load data points onto the map for seven different scales of green tide data. The experiment aimed to compare the data loading time or structure construction time of each method with the total data loading time. The experimental results are presented in Table 1 and Fig 2.

**Table 1.** Comparison of time for building and loading data in spatial indexing structures

| Data | Data level | Number of data points | Direct loading/s | Quadtree/s | R-tree /s | KD-tree/s: |
|---|---|---|---|---|---|---|
| 20220607 | Thousand-level | 3394 | 0.0699 | 0.1011 | 0.2002 | 0.0187 |
| 20190521 | | 6956 | 0.1299 | 0.2310 | 0.5807 | 0.0480 |
| 20200527 | ten thousand-level | 45111 | 0.9716 | 1.5825 | 20.7891 | 0.2711 |
| 20200621 | | 50398 | 1.1452 | 1.8454 | 26.2344 | 0.3063 |
| 20190602 | | 157121 | 3.9220 | 5.7163 | 238.8946 | 1.0443 |
| 20190611 | hundred thousand-level | 582670 | 13.1197 | 23.1420 | >800 | 3.8247 |
| 20190623 | | 853868 | 20.1212 | 31.5116 | >1600s | 5.4819 |

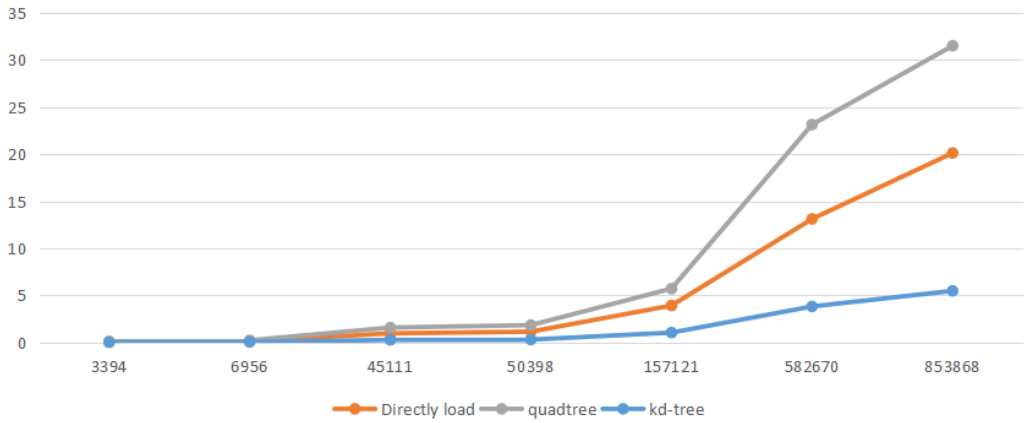


Fig 2 Comparison of time for constructing and loading data in spatial index structures

From the experimental data comparison in Table 1, it can be observed that quadtree loading takes even longer than direct loading without any processing, failing to improve the performance of visualization loading. Due to its overly complex construction process, when the data scale reaches tens of thousands, the construction time of the R-tree data model exceeds 20 seconds. When the scale reaches hundreds of thousands, it leads to extremely long construction time, causing program lag or even system crashes. When KD-tree is used as the spatial indexing structure, compared to direct loading or constructing other types of spatial indexing structures, the total time of construction and loading is the shortest.

From the line comparison in Fig 2, it can be observed that the performance advantage of using KD-tree as a spatial indexing structure becomes more prominent as the quantity of data increases. Even when dealing with green tide data at the scale of millions, it is able to rapidly construct the structure and achieve fast data loading. Through the comparative experiments on spatial indexing structures, the rationality and efficiency of using KD-tree for constructing storage structures have been validated.

## 4.3. Optimization Experiment of KD-Tree Construction Algorithm based on LSH Density Estimation

After selecting three sets of data with different distribution types but the same quantity, the data storage construction algorithm before and after the upgrade was applied respectively for construction experiments. The construction time and loading time were measured and compared. To facilitate comparison, the experimental results were rounded to four decimal places.

**Table 2.** Performance comparison of optimized KD-tree construction algorithms based on data density partitioning.

| Data | Data distribution type | Number of data points | Loading time in seconds for the original KD-tree construction algorithm/s | Loading time in seconds for the density-based partitioning optimized KD-tree construction algorithm/s |
|---|---|---|---|---|
| 20220607 | Scattered | 3394 | 0.0187 | 0.0172 |
| 20230515 | Dense | 1326 | 0.0080 | 0.0070 |
| 20200527 | Scattered | 45111 | 0.2711 | 0.2396 |
| 20200621 | Dense | 50398 | 0.3063 | 0.2963 |
| 20230705 | Scattered | 379887 | 2.4624 | 1.9653 |
| 20230622 | Dense | 491416 | 3.1546 | 3.0724 |

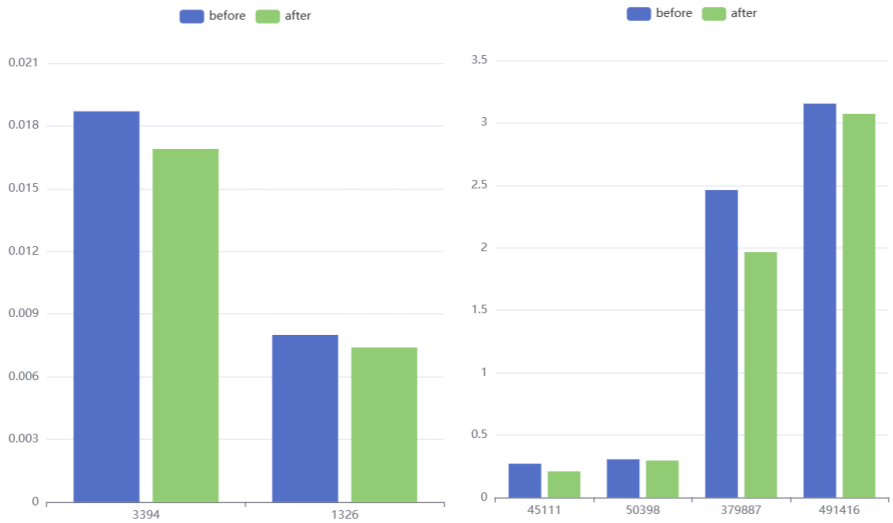


**Fig .3** Performance comparison of optimized KD-tree construction algorithm based on data density partitioning

As shown in Table 2 and Fig 3, the experimental results indicate that by applying the optimized KD-tree construction

algorithm based on data density partitioning, both the construction time and data loading time are reduced compared to the original KD-tree construction algorithm when dealing with densely distributed and scattered distributed data.

Performance analysis based on Fig 3 reveals that, compared to densely distributed data, when facing scattered green tide data, utilizing the optimized data construction algorithm based on density estimation for constructing the data storage structure can achieve significant performance improvements on top of the original construction algorithm. However, the improvement effect is relatively small when dealing with densely distributed data, with minimal noticeable enhancements.

# 5. Conclusion

In this paper, an innovative algorithm for constructing spatial vector data storage based on KD-tree and density estimation is proposed. The optimization of KD-tree construction is achieved by introducing Local Sensitive Hashing (LSH) for density estimation. Firstly, experimental comparisons validate that the KD-tree data storage structure enables efficient storage of spatial vector data. By selecting partition dimensions and partition thresholds, fast retrieval and querying of data are achieved, leading to significant advantages in terms of storage space utilization and query response time

Furthermore, the density estimation of data distribution based on the Local Sensitive Hashing (LSH) algorithm is employed to optimize the KD-tree construction algorithm, enabling faster construction of spatial index structures. By introducing LSH techniques for density estimation, the efficiency of density estimation is improved through LSH hashing calculations using MinHash signatures of data points. The optimization algorithm takes into account the distribution characteristics of data points, particularly in handling unevenly distributed data, resulting in more efficient storage and querying. The introduction of this algorithm not only provides new technical means for efficient management of spatial vector data, but also offers strong support for spatial data processing and analysis in fields such as geographic information systems, environmental monitoring, and urban planning.

# Acknowledgements

This paper was financially supported by The HY-1C/D satellite data was obtained from the website: https://osdds.nsoas.org.cn. The authors of this paper would like to thank the National Satellite Ocean Application Center for providing data support.

# References

[1] Vo H T, Bronson J, Summa B, et al.Parallel visualization on large clusters using MapReduce[C]//Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on IEEE,2011:81-88.

[2] Yang X, Liu S, Feng K, et al. Visualization and adaptive subsetting of earth science data in HDFS: A novel data analysis strategy with hadoop and spark[C]//Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computingand Communications (SustainCom) (BDCloud-SocialCom-SustainCom), 2016 IEEE International Conferences on IEEE2016:89-96.

[3] Eldawy A, Mokbel M F, Jonathan C. Hadoop Viz:A MapReduce framework for extensible visualization of big spatial data[C]//Data Engineering (ICDE), 2016 IEEE 32nd International Conference on IEEE, 016601-612.