International Journal of Geo-Information

MDPI

Article

Integrating NoSQL, Hilbert Curve, and R*-Tree to Efficiently Manage Mobile LiDAR Point Cloud Data

Yuqi Yang [1], Xiaoqing Zuo [1,*], Kang Zhao [2] and Yongfa Li [1]

[1] Institute of Land and Resources Engineering, Kunming University of Science and Technology, Kunming 650093, China; yyq0730@stu.kust.edu.cn (Y.Y.); yfli@stu.kust.edu.cn (Y.L.)
[2] Department of Natural Resources of Yunnan Province, Kunming 650224, China; kzhao@whu.edu.cn
* Correspondence: zxq@kust.edu.cn

Abstract: The widespread use of Light Detection and Ranging (LiDAR) technology has led to a surge in three-dimensional point cloud data; although, it also poses challenges in terms of data storage and indexing. Efficient storage and management of LiDAR data are prerequisites for data processing and analysis for various LiDAR-based scientific applications. Traditional relational database management systems and centralized file storage struggle to meet the storage, scaling, and specific query requirements of massive point cloud data. However, NoSQL databases, known for their scalability, speed, and cost-effectiveness, provide a viable solution. In this study, a 3D point cloud indexing strategy for mobile LiDAR point cloud data that integrates Hilbert curves, R*-trees, and B$^+$-trees was proposed to support MongoDB-based point cloud storage and querying from the following aspects: (1) partitioning the point cloud using an adaptive space partitioning strategy to improve the I/O efficiency and ensure data locality; (2) encoding partitions using Hilbert curves to construct global indices; (3) constructing local indexes (R*-trees) for each point cloud partition so that MongoDB can natively support indexing of point cloud data; and (4) a MongoDB-oriented storage structure design based on a hierarchical indexing structure. We evaluated the efficacy of chunked point cloud data storage with MongoDB for spatial querying and found that the proposed storage strategy provides higher data encoding, index construction and retrieval speeds, and more scalable storage structures to support efficient point cloud spatial query processing compared to many mainstream point cloud indexing strategies and database systems.

Keywords: point cloud data; MongoDB; Hilbert curve; R*-tree; spatial index; LiDAR

Citation: Yang, Y.; Zuo, X.; Zhao, K.; Li, Y. Integrating NoSQL, Hilbert Curve, and R*-Tree to Efficiently Manage Mobile LiDAR Point Cloud Data. *ISPRS Int. J. Geo-Inf.* **2024**, *13*, 253. https://doi.org/10.3390/ijgi13070253

Academic Editors: Wolfgang Kainz and Eliseo Clementini

Received: 5 June 2024
Revised: 9 July 2024
Accepted: 12 July 2024
Published: 14 July 2024

1. Introduction

Light Detection and Ranging (LiDAR) technology originates from the 1960s and was incorporated into airborne platforms in the 1980s [1]. Currently, LiDAR is used to gather extensive spatial data of three-dimensional (3D) geospatial objects by combining multi-platform construction methods with high-precision acquisition equipment and serves a variety of decision-making processes by involving real-world spatial data [2,3]. With advances in 3D laser scanning technology, point clouds have become the third largest spatiotemporal data source in geospatial applications after vector maps and images [4]. In addition, 3D scanning, as one of the four major branches of the traditional geospatial industry (global navigation satellite systems (GNSS) and positioning, geographic information system (GIS) and spatial analysis, Earth observation, and 3D scanning), will become the fastest-growing market and lead to the rapid development of smart cities, intelligent transportation, global mapping, and other fields [3–5].

Point clouds have high value in research and for applications in many fields, such as geographic information science. Point clouds can accurately depict the 3D morphological structure of vegetation, glaciers, and islands, thus providing important support for global forest accumulation, biomass estimation, global glacier material balance, marine economic

development management, and sea defense security. In smart cities, point clouds play an increasingly important role in urban refinement management, urban security analysis, and 3D change detection. In intelligent transportation, point clouds can realize real-time motion target detection and positioning, real-time obstacle avoidance, core support for HD map production, accurate and intuitive 3D location information, and precise path planning and control strategies beyond the capacity of sensors. Therefore, many countries are collecting LiDAR data in large quantities, thus accumulating a large number of datasets [5,6]. The rapid generation of hundreds of millions of unstructured data points with rich attribute information in a short time leads to common problems associated with the increasing size, density, and complexity of the data [7]. In addition, LiDAR data are typically stored, shared, and exchanged using LAS/LAZ format files, which are inefficient and poorly scalable when dealing with massive datasets [5,8]. Therefore, databases, which allow for centralized access, concurrent retrieval, distributed storage, and indexing support are more advantageous in managing large LiDAR data [8].

With the popularization and application of high-resolution vehicle-mounted laser scanning systems, the rapid processing of large amounts of scattered point cloud data has become the focus of international research. The post-processing of point cloud data, such as simplified filtering, semantic segmentation, feature extraction, and other interactive operations, is limited by the performance of data management, which greatly restricts the rapid accessibility of point cloud data for comprehensive applications [9]. When utilizing an out-of-core approach for massive point cloud management, achieving efficient data loading, retrieval, and scheduling using an index structure is necessary [10]. On the one hand, a single index structure shows high performance when presented with a small amount of uniformly distributed data but is not appropriate for large-scale discrete point sets. On the other hand, a complex nested index structure can integrate the advantages of different indexes to organize a large number of 3D points but presents difficulties in the construction and maintenance of indexes. Additionally, the discrete and unstructured nature of point cloud data renders local file storage inadequate for network parallel computing and business needs [11].

An increasing number of studies have been devoted to the development of efficient point cloud data management (PCDM) systems. While some of these systems utilize file-based techniques for storing and querying point cloud data, a significant number aspire to develop PCDM systems that rely on database technologies. The focus of PCDM research is on coping with large amounts of heterogeneous point cloud data. Therefore, PCDM solutions that rely on database technologies are primarily focused on achieving greater scalability while maintaining acceptable performance levels. Traditional Relational Database Management Systems (RDBMSs), designed primarily for structured data, experience performance degradation when dealing with unstructured storage [12]. Research on distributed storage for point clouds has focused on HDFS [1,13–15] and MongoDB [11,16,17], whereas relatively few studies have utilized HBase [7] to manage point cloud data. Notably, systems built on HBase have the flexibility to scale to a large number of nodes for accommodating complex data, displaying greater fault tolerance and scalability [18]. Consequently, the implementation of efficient point cloud data storage, retrieval, and indexing strategies in NoSQL databases represents a popular research direction.

Despite the remarkable success in exploring the efficient management of large LiDAR datasets supported with NoSQL, some problems remain. First, the spatial distribution characteristics of point clouds are not fully considered; second, the storage system cannot be integrated with high-performance spatial indices, which makes data transfer and storage very inefficient. To overcome these problems, this paper proposes a 3D point cloud indexing strategy based on the massive spatial information existing in the point cloud itself and the discrete nature of points in 3D space, and it integrates Hilbert curves, R*-trees, and B^+-trees and uses MongoDB to complete the storage of unstructured mobile LiDAR point clouds and realize the efficient organization and fast retrieval of large point cloud datasets. This hybrid indexing model can fully utilize the spatial information in the point cloud data

and exhibits better query performance in comparison with both the existing single index structure and the composite index structure. At the same time, it utilizes the advantages of in-memory computing to improve the efficiency of retrieving data from secondary storage and improves the storage and querying applications of large-scale point cloud data in an NoSQL environment.

Section 2 provides an overview of the data structures and storage management systems used to manage large-scale point cloud datasets; Section 3 details the proposed hierarchical indexing architecture and its implementation; Section 4 evaluates the feasibility of the proposed strategy by comparing this paper's approach with other point cloud indexing and storage schemes; and Section 5 summarizes this study and discusses the limitations and future work.

2. Related Work

This section will provide current research on point cloud spatial indexing and point cloud storage management systems.

2.1. Indexes for Point Cloud Data

Overall, this study aims to increase the flexibility and maneuverability of point clouds for subsequent processing by building reliable and efficient spatial indexes [19]. As depicted in Figure 1, the point cloud indexing structures that have received the most attention are mainly based on regular lattice grids, Quadtree, Octree, R-tree and its variants, and KD-tree [8].

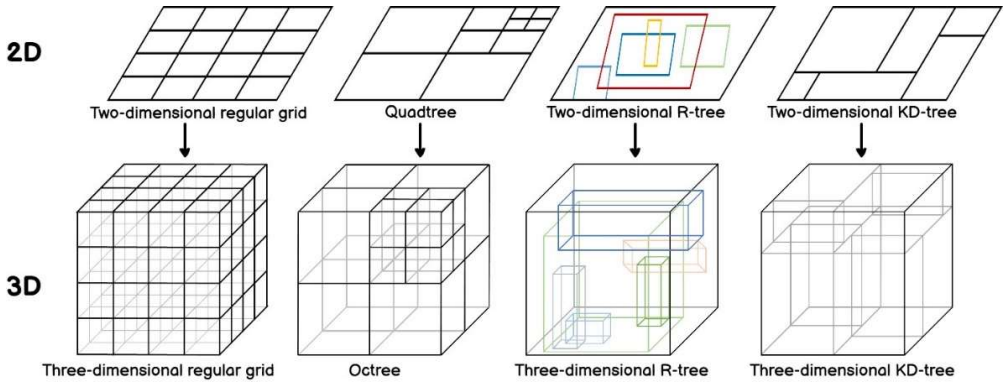

Figure 1. Common spatial data structures.

Kim et al. [20] implemented point cloud storage and indexing through the Discrete Global Grid System (DGGS) with PH trees to support efficient point cloud spatial queries. Space-filling curves have been introduced in some studies [21–24] to encode the mesh and realize the downscaling of the point cloud. The use of efficient 1D queries to reduce the number of 3D queries performed improves the query and processing efficiency of the point cloud to some extent. This grid-based indexing structure can complete the organization of several point clouds in a short period of time but faces difficulties associated with accuracy limitations, data imbalance, grid division complexity, and data update extension. Moreover, the setting of the curve order also needs to be fully considered. A lower order is not meaningful for data division, and performance improvement is not obvious; a higher order requires a lot of time for index construction, and the query statement will become large and complex with the expansion of the query range. Even if the concept of spatial adaptive partitioning is introduced, the limited curve accuracy will make the results filtered out by the index contain a lot of "fake data", which requires a lot of time to refine. Octrees and Quadtrees are flexible extensions of grid indexes with simple implementation, high automation, and a wide range of applications, making them ideal for managing hundreds of millions of point cloud data [25]. Tian et al. [26] reduced blank space by reconstructing the minimum boundary rectangle of the Octree's child nodes, which effectively solved the

imbalance problem of the Octree caused by the uneven distribution of the point cloud space and improved query efficiency. Huang [27] addressed the data redundancy problem of the traditional multi-resolution point cloud structure by storing point cloud data with different resolutions in leaf nodes at different levels of the Octree. However, with increasing data size, massive discrete point clouds will inevitably lead to excessive depth of the Octree, which remarkably degrades query efficiency. KD-trees have also been used to spatially partition the point cloud and organize it through each partition of an Octree to achieve fast indexing of massive discrete point clouds [19,28]. R-trees offer superior spatial query efficiency compared to Octrees and KD-trees [29]. Zhu et al. [30] implemented a clustering algorithm (K-means) to optimize the spatial clustering grouping and insertion of R-trees to reduce node overlaps and search paths in R-trees. Gong et al. [31] proposed the 3DOR-tree, which uses Octree for fine-grained spatial partitioning of point clouds and organizes Octree leaf nodes as R-tree root nodes for increased spatial query efficiency. Wang et al. [32] proposed the 3DOR*-tree based on 3DOR-tree, which utilizes R*-tree to organize the leaf nodes of the Octree to provide superior point cloud query performance. Yu et al. [33] divided the data into grids based on the spatial distribution of point clouds and used R-trees to manage the non-empty grids. In this solution, the point clouds within the grid are organized via Octrees and Quadtrees, which achieves efficient management of massive point clouds. However, these complex and efficient indexing structures are designed with little consideration for integration with database storage systems; thus, massive amounts of point cloud data cannot be simultaneously loaded into memory for processing. Therefore, point cloud storage management systems supported with composite index structures must be explored further.

2.2. Storage Management System for Point Cloud Data

In the field of storage and management of point clouds, there have been numerous studies based on RDBMS and NoSQL. Deibe, Amor, and Doallo [34] and Juan A. Béjar-Martos et al. [8] conducted a comprehensive analysis of the most mature and widely adopted RDBMS and distributed storage technologies. They concluded that MongoDB and Cassandra exhibit superior performance in managing extensive point cloud datasets. The authors of [11,16] harnessed the automatic slicing mechanism of MongoDB to proficiently handle large point cloud datasets, yielding promising results. However, the usefulness of these methods is limited to file selection, since data management is only performed at the file abstraction level. Additionally, discussions concerning LiDAR data within a Hadoop framework have revolved around the performance of distributed storage and parallel computing [1,13–15]. Since MapReduce only supports HDFS as a data source and lacks real-time data processing capabilities, data transfer and disk I/O can become a performance bottleneck for very large quantities of point cloud data. Pajić et al. [35] proposed a Spark-based point cloud storage model that ingeniously combines space-filling curves and HBase for efficient organization and management of vast point cloud datasets. Deibe, Amor, and Doallo [36] stored the point cloud uniformly in each node of the Cassandra cluster with scheduling and processing via Spark. Vo et al. [6] proposed Ariadne3D based on HBase and Spark, which manages massive point clouds through flexible and scalable data encoding, indexing mechanisms, and resource scheduling. Rueda-Ruiz et al. [17] proposed the conceptual SPSLiDAR model to support efficient point cloud retrieval and concurrent access through MongoDB with Octree. However, almost all of these point cloud management systems organize point clouds through a single data structure, such as Octree, R-tree, and space-filling curves. When facing large-scale discrete point cloud data, the maintenance, updating, and querying of these index structures can become extremely complex, thereby reducing the efficiency of point cloud management. Therefore, the composite index structure should be integrated with current mainstream NoSQL systems to realize the efficient management of discrete point clouds by combining the query efficiency of composite indexes with the flexible and scalable storage structure of NoSQL.

Thus, in this study, we aim to explore the point cloud management strategy supported with MongoDB with composite index structure to address the drawbacks associated with using a non-relational database with a single index structure for storing and indexing point clouds. The proposed point cloud data storage method based on the MongoDB database was implemented to preserve the spatial distribution characteristics of point clouds and provide efficient point cloud query support.

3. Methodology

This section describes the methods. Then, we will introduce the hierarchical index tree in detail, as well as the MongoDB storage structure design and spatial query processing based on the hierarchical index tree.

3.1. The Architecture of the Hierarchical Index

To provide efficient point cloud data query support, a hierarchical index structure is proposed, and it includes a (i) global index, in which a Hilbert tree is constructed to globally identify all the point cloud partitions and a (ii) local index, in which an R*-tree is utilized to index 3D point objects in the partitions to provide spatial query support for point clouds. The hierarchical index is used for query task scheduling to support NoSQL-based point cloud spatial queries.

3.1.1. Global Index for High Data Locality

A common method for pruning the query task, reducing the I/O seek time, and improving the retrieval efficiency is to construct a global index by partitioning the space. Meanwhile, the spatial association of spatially partitioned objects is maintained using the 3D points in space.

In particular, tile-based structures are widely used to store and manage large-scale point cloud data. In a slice-based structure, the raw point cloud data are divided into many slices (usually squares) based on their spatial distribution, and each slice is organized into an LAS file [15]. This "divide and conquer" aspect of data partitioning can divide big data into relatively small independent sub-blocks, thereby improving data processing and computation and increasing data storage and management system efficiency [37]. However, this uniform partitioning strategy is more effective when applied to terrestrial LiDAR systems, whereas the discrete spatial distribution characteristics of the point cloud need to be considered for the more widely used mobile LiDAR scanning systems to avoid data skewing after division [38,39].

Moreover, instead of partitioning the point cloud in a spatially homogeneous manner, these data are divided into spatial regions that do not overlap and can be dynamically adjusted by setting a threshold. In addition, the partitions are encoded using space-filling curves to realize the reduced dimensional spatial representation of the point cloud for subsequent data retrieval and access.

A space-filling curve is a one-dimensional continuous curve that can run through multi-dimensional space, and it represents an effective method of dimensionality reduction that is widely used in various types of GIS algorithms [40]. Commonly used space filling curves include the Z curve and Hilbert curve [41]. The Z curve has local proximity, although it also has serious spatial mutability, i.e., the points coded by neighboring numbers may not be adjacent to each other, and its coding cannot effectively reflect spatial distance [42]. The Hilbert curve has optimal spatial aggregation and discrete approximation abilities, and adjacent points in space are adjacent and continuous on the curve; thus, it can realize the mapping from multidimensional space to one-dimensional space well [40,43]. Therefore, the Hilbert curve is chosen as the basis of the coding algorithm in this study and realizes the adaptive grading of discrete data by setting the following threshold: higher-order Hilbert curves are generated in regions with dense 3D points, and lower-order Hilbert curves are generated in regions with sparse 3D points. The underlying coding algorithm for Hilbert curves is available on GitHub [44]. To clearly describe the process of spatial adaptive

segmentation, a two-dimensional example is provided in Figure 2. The subsequent plots in this paper are likewise provided in two dimensions.

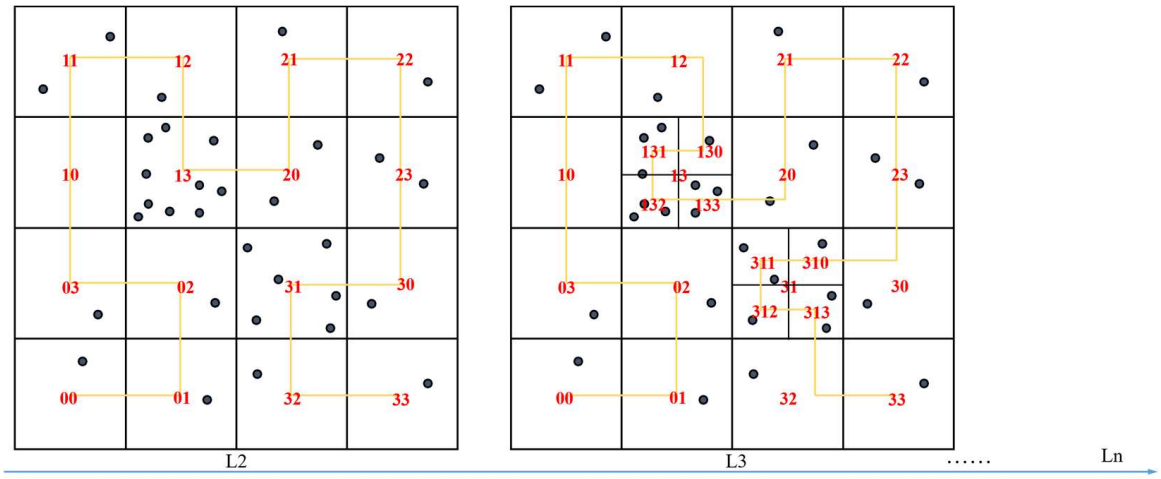

Figure 2. Hilbert coding strategies based on inhomogeneous space partitioning.

The Hilbert mesh is dynamically divided via the iterative octet method of 3D spatial mesh. At the first division, based on the set initial Hilbert curve order (N_{ini}), the three-dimensional space is uniformly divided into a $2^{Nini} \times 2^{Nini} \times 2^{Nini}$ grid. Subsequently, the amount of data contained within each grid is counted and the grid cells are filtered for iterative subdivision according to a set threshold for region delineation (P_{max}) to construct locally finer grids. Subdivision stops when the subdivision order reaches the predefined maximum order (N_{max}) or when the point data in a single grid cell are less than the threshold value P_{max}. The initial Hilbert curve is determined with the following equation:

$$N_{ini} = \left[\log_8 \frac{P_{all}}{P_{\max}} + 1.5 \right] \quad (1)$$

where N_{ini} is the initial coding level, P_{all} is the total number of points, and P_{max} is the region splitting threshold. In general, p_{max} can be selected based on hardware performance and network speed or the size of the most frequently queried study area. For most applications, a few thousands to tens of thousands of points is an appropriate value size. For example, for 1 million points and a splitting threshold of 2000, there will be 500 records if the data points are uniformly distributed in 3D space. A level 3 Hilbert curve can pass through 512 grid cells; therefore, this value is sufficient. However, due to the uneven distribution of data points, we chose level 4 as the initial coding level because level 4 can pass through 4096 grid cells. Choosing the appropriate initial coding level based on the distribution of points in 3D space can reduce the time spent on the subsequent refinement and combination process.

Spatially adaptive partitioning is used as the basis for constructing the Hilbert tree, which enables the construction of the global index (Figure 3). The unique corresponding Hilbert code of the point cloud partition indicates its position in the Hilbert tree in the storage system. Additionally, to avoid redundant coding calculations and grid space, we optimized the index structure of the Hilbert tree and proposed a compact Hilbert tree. For the eight leaf nodes obtained from the subdivision, the leaf nodes that do not contain data are chosen to be eliminated from the index tree and the minimum bounding box that can contain them is obtained by re-computing according to the remaining leaf nodes to complete the reconstruction of the parent node.

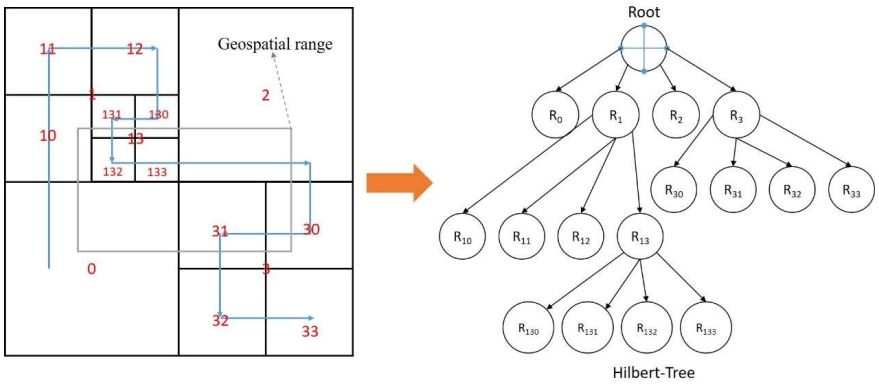

Figure 3. Overview of Hilbert tree.

3.1.2. Local Index of Point Clouds Based on R*-Tree

R*-tree optimizes the data insertion and node splitting logic by introducing a series of measures for regions, boundary shapes, and overlapping nodes and proposes a forced reinsertion mechanism, which makes the tree structure more reasonable and efficient. The memory persistence feature of R*-tree makes it possible to retrieve the objects without accessing the actual data objects and provides a list of the objects directly in line with the spatial relationship, which lowers the response latency. Due to the fixed form of index elements, the volume of the R*-tree is proportional to the number of indexed objects. Under conditions of large data volumes, the construction of a single centralized R*-tree often leads to a larger depth of the tree and a high volume of the index, index loading, memory cost traversal, and time cost. Variants of the R*-tree are proposed as a distributed index for processing multidimensional spatiotemporal data in a cluster of workstations [45]. Therefore, we construct an R*-tree [46] as a local index for each point cloud partition while controlling the index volume. The implementation of R*-tree can be referred to GitHub [47], which provides ideas for the design of the algorithm in this paper. Readers can modify and extend it according to their needs. When performing a spatial query, the corresponding local index is loaded into memory on demand, and data retrieval is performed based on the input spatial query range.

R*-tree is utilized to store the relevant information of the point objects in point cloud blocks, i.e., each leaf node of the R*-tree stores the attribute information of the 3D point objects, and the non-leaf nodes store their corresponding minimum bounding box (MBB) as well as the pointers to their child nodes.

As shown in Figure 4, the R*-tree comprehensively optimizes the volume, edge length and degree of superposition of each MBB in the path to achieve more reasonable spatial clustering. It offers performance advantages over R-tree in terms of rectangular data, multidimensional point data, and query operation and map overlay display. In practical applications, although the realization cost of R*-tree is slightly higher than that of R-tree, the point cloud is processed in chunks and the number of three-dimensional points indexed by a single R*-tree is not excessive. Therefore, the complexity of constructing and maintaining R*-tree is not significantly higher than that of R-tree. In addition, comparisons with other common point cloud data structures, such as Octree and KD-tree, reveal that R*-tree is much more complicated to construct and maintain. However, with its excellent node splitting design and internal index structure, R*-tree provides more efficient data retrieval support.

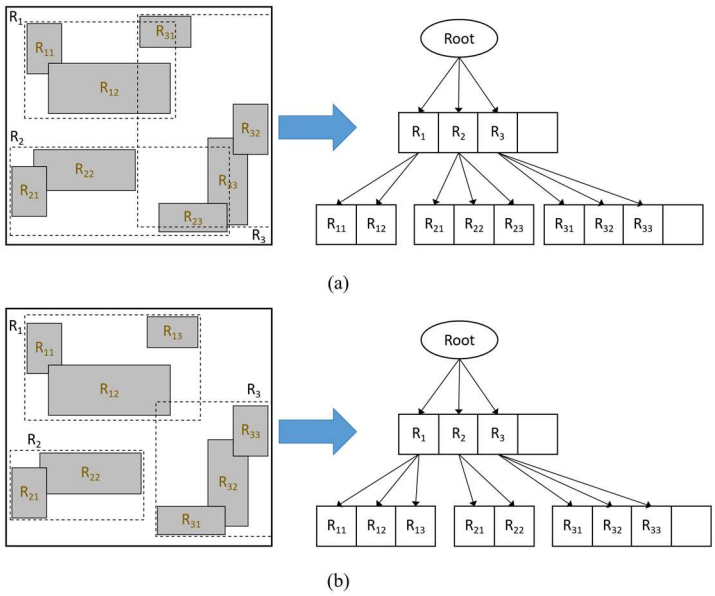

Figure 4. Comparison of R-tree and R*-tree. (**a**) R-tree and (**b**) R*-tree.

3.2. Construction of Hierarchical Index Tree

The hierarchical index structure is organized in the form of nested secondary index trees, in which the Hilbert tree is the primary data structure, and the 3D R*-tree is the secondary data structure. This nested tree structure can fully exploit the fast convergence of the Hilbert tree to quickly divide the three-dimensional space; however, it also has the potential advantage of the R*-tree for efficient retrieval in high-dimensional space. In this section, we will describe in detail how to integrate both processes. Figure 5 shows the organization of the integrated data structure of Octree and 3D R*-tree, and the leaf nodes of the Hilbert tree represented using dashed boxes indicate redundant nodes without data that need to be eliminated.

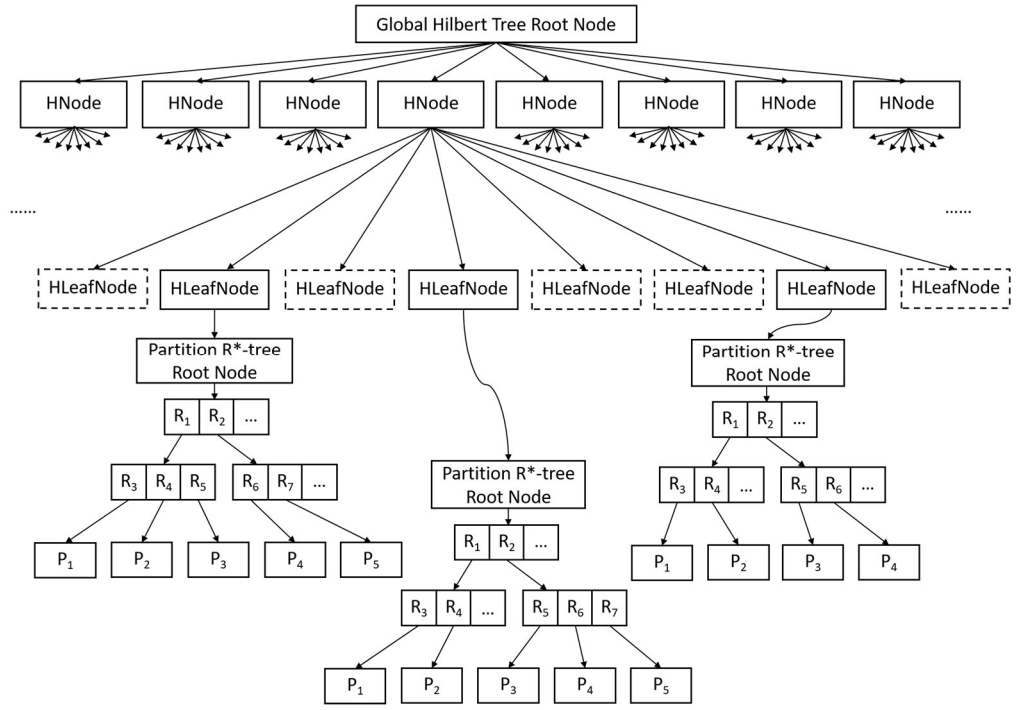

Figure 5. Data structure organization of Hilbert tree integrating with 3D R*-tree.

The construction of a hierarchical index tree involves two sub-processes: spatial partitioning of a Hilbert tree and spatial object storage of a 3D R*-tree. The construction algorithm integrates the two sub-processes together and associates these processes through the leaf nodes of the Hilbert tree. In the spatial division process of the Hilbert tree, the leaf nodes of the Hilbert tree that satisfy the threshold condition are used as the root nodes of the 3D R*-tree. Based on the given 3D R*-tree fan-out parameters (each node is allowed to contain the maximum number of entries and the minimum number of entries, f_{max} and f_{min}), spatial objects are inserted individually to construct the internal nodes of the 3D R*-tree until all spatial objects have completed the insertion operation, that is, the construction of the 3D R*-tree is completed. The specific construction process is shown in Figure 6.

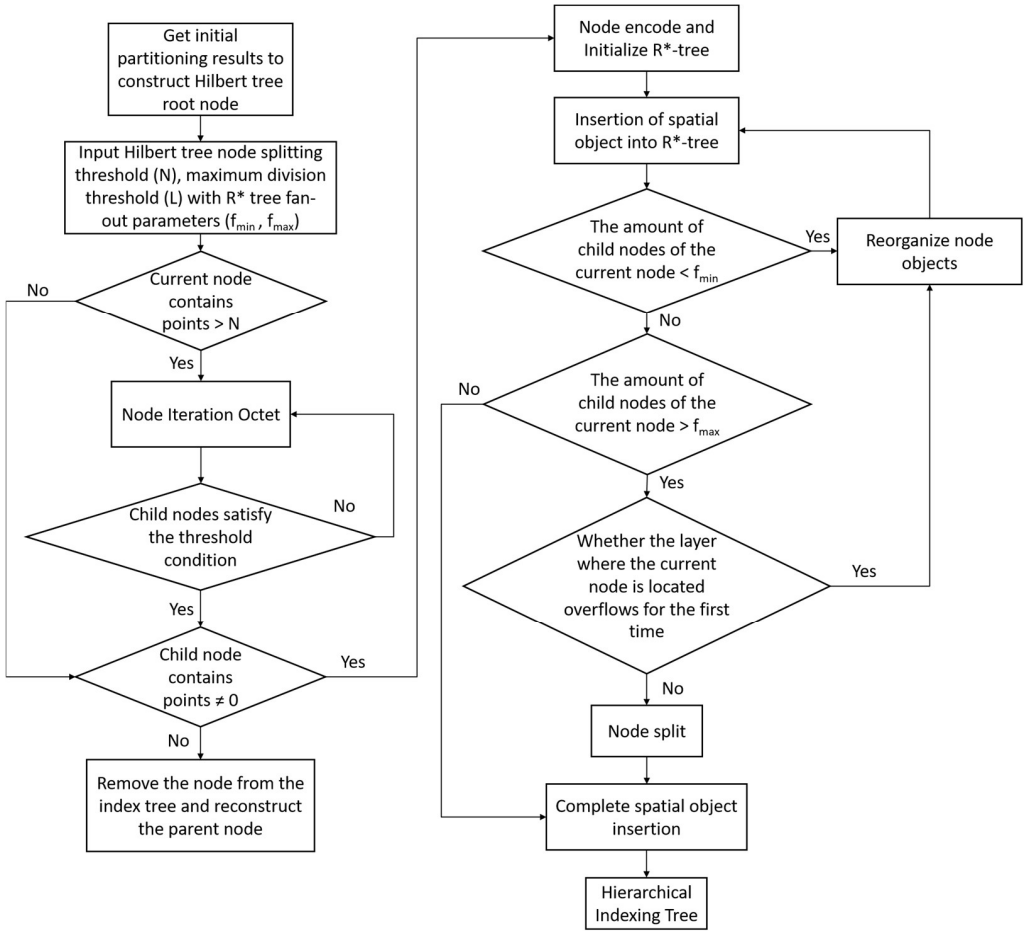

Figure 6. Construction flow chart of hierarchical index tree.

The hierarchical index tree construction algorithm is described as follows:

Algorithm Input: Hilbert tree split threshold (N) and R*-tree fan-out parameters (f_{min}, f_{max});

Algorithm Output: hierarchical index tree structure;

- Step 1: Obtain the initial partitioning results, take each independent partition as a child node of the Hilbert tree, and the smallest bounding box containing all child nodes is the root node;
- Step 2: Count the number of points ($ptNum$) contained in each child node; if $ptNum > N$, divide the space into eight child nodes uniformly and assign the 3D point objects to the corresponding child nodes. If $ptNum \leq N$ is satisfied, the division stops and goes directly to step 3; otherwise, the recursive division needs to be continued;
- Step 3: Construct the initialized three-dimensional R*-tree for leaf nodes of the Hilbert tree that satisfy the threshold condition if $ptNum \neq 0$ in the current node, and in-

sert the three-dimensional point objects into the three-dimensional R*-tree one by one; otherwise, remove the current node from the index tree and reconstruct the parent node;

- Step 4: Perform the insertion operation of the 3D R*-tree; if the inserted node contains the number of child nodes ($chNum$) < f_{min} after insertion, then this reorganizes the node objects within the node; otherwise, continue to step 5;
- Step 5: Divide an overflow situation after the insertion of the node containing $chNum > f_{max}$ into two cases. If the node is in the layer of the first overflow, then perform the re-insertion operation; otherwise, perform the node split operation. If $chNum < f_{max}$, then this three-dimensional point object is used to complete the insertion and continue on to step 6;
- Step 6: Check whether 3D point objects have not been inserted; if so, repeat step 4 and step 5 until all 3D point objects are inserted into the tree structure and the algorithm ends.

3.3. Point Cloud Storage and Query

In this paper, by refining the spatial units, the dense spatial point cloud data are divided into relatively small groups to participate in the point cloud hierarchical index construction, and a spatial index framework for large-scale point cloud data oriented to NoSQL is designed based on the hierarchical index structure (see Figure 7). The main idea is to organize the point cloud by spatial encoding via R*-tree and accelerate point cloud spatial query processing via one-dimensional B^+-tree. The spatial query uses the classical two-stage processing strategy. The index screening stage is based on the global Hilbert tree for rough querying, according to which the R*-tree of the corresponding partition is obtained for precise query to obtain the final result.

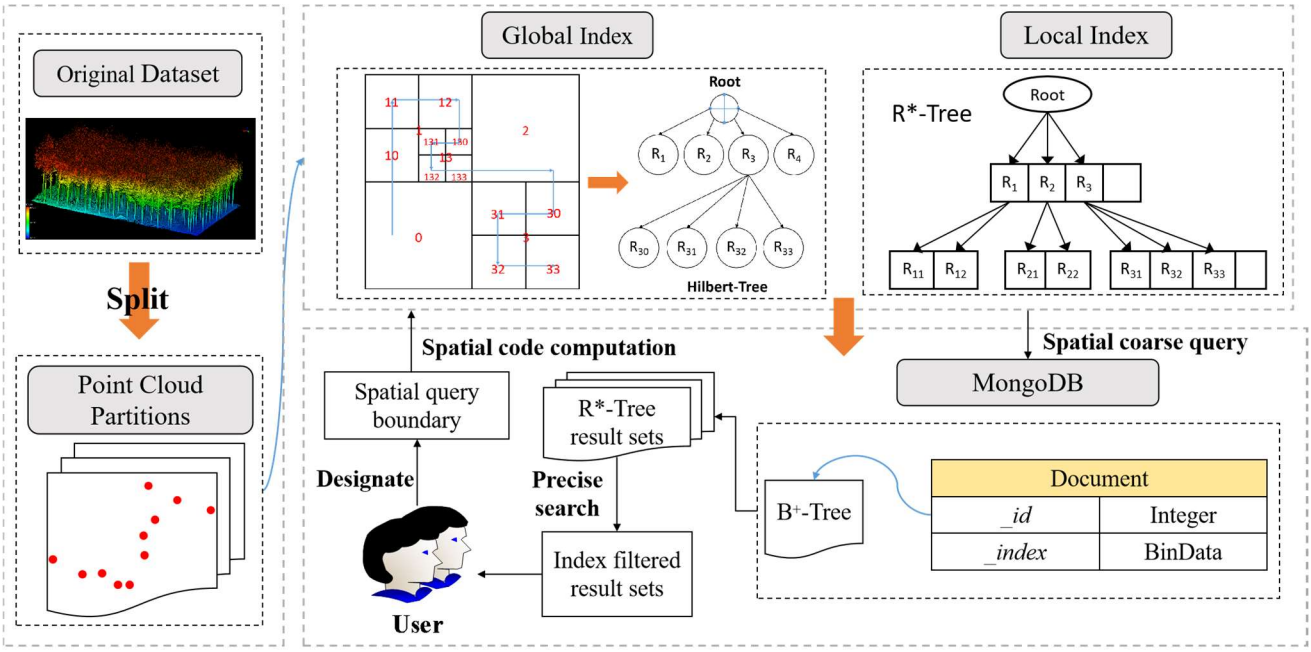

Figure 7. Overview of the point cloud storage and indexing framework.

3.3.1. Structural Design of MongoDB

Spatial query processing of large-scale data in file systems without native index support usually requires traversing all spatial information records to match the corresponding query results; thus, the query efficiency is extremely low. However, in mature spatial databases, R-Tree indexes are usually used to manage point cloud data. In addition, because of the exponential increase in the number of mobile application users,

discrete spatial data management methods require cloud storage support. In the era of big data, the application of NoSQL is becoming increasingly widespread. Therefore, in this paper, the document-based non-relational database MongoDB is used as the storage medium to implement the hierarchical index. The organizational structure of MongoDB is database–collections–document objects–elements; collections are similar to tables in relational databases, document objects are similar to records, and elements are similar to fields. MongoDB supports record collections that do not require the same strict structure, presents flexible design patterns, and has a rich data format that is suitable for point cloud data storage. We designed the MongoDB-oriented point cloud document storage structure based on the hierarchical index structure, as shown in Figure 8. This section will focus on the MongoDB-based point cloud storage scheme.

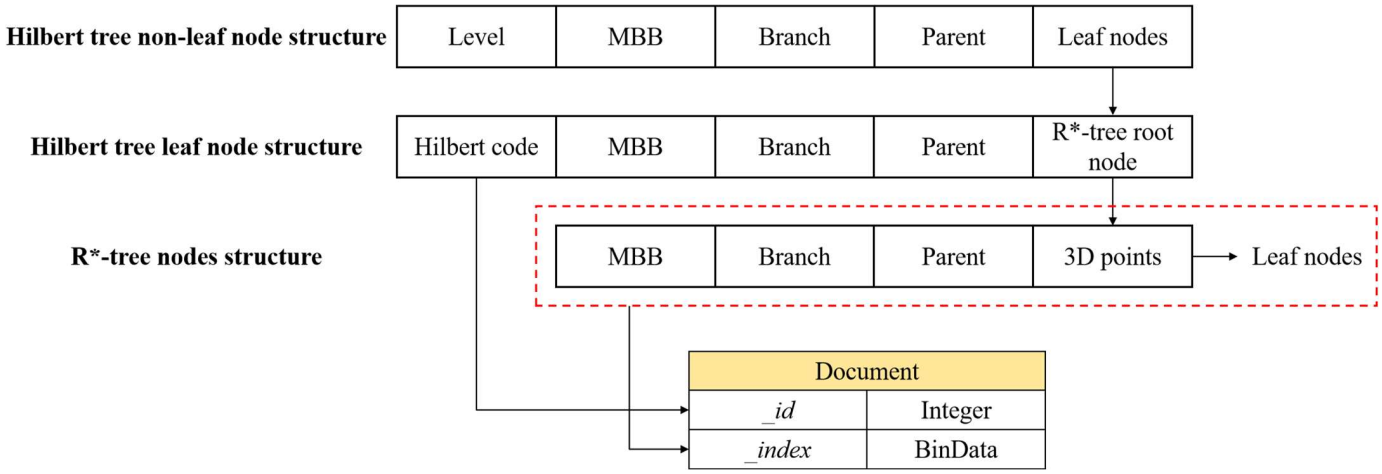

Figure 8. Document storage model based on hierarchical index structure.

The global index is designed to index all partitions and their physical locations to obtain a global overview of the point cloud partitions in the storage system. Since the global index resides in the memory of a single node, it should take up as little memory space as possible and be easily implemented. Therefore, the Hilbert tree with its leaf nodes only holds the Hilbert-encoded information of the corresponding grid, indicating its position in the storage system with the Hilbert tree, and does not actually store the point cloud data.

For the Hilbert tree leaf nodes in which the actual data are stored, the local index R*-tree is stored in the collection and the documents within the collection store the ROWID number of the root node of the R*-tree and its node information. The ROWID of the root node is the unique identifier of the root node document, and it is used to read the root node data from the database as well as the data for any node in the R*-tree. In this study, Hilbert encoding is chosen as the ROWID of each document in MongoDB, which represents an important foundation for MongoDB to realize distributed storage. To facilitate point cloud spatial querying, this study constructs a B^+-tree index on the ROWID field and ensures that the nodes of a single R*-tree that uniquely correspond to the grid Hilbert encoding are also deposited into the same set as the elements of the document. Moreover, the nodes are divided into leaf and non-leaf nodes, which have different structures. For storage convenience, this study serializes the node data of the R*-tree as binary blocks, that is, BinData type elements deposited into the document. Among them, the leaf nodes record the attribute information of the point cloud, the non-leaf nodes store the corresponding minimum bounding boxes and pointers to child nodes, and access to non-leaf nodes reveals whether the child nodes satisfy the coarse check requirements.

3.3.2. Point Cloud Spatial Query

Querying spatial data is the inverse process of spatial indexing and data storage and is closely related to the data storage model and index structure. Spatial-oriented queries are

query operations based on spatial indices, and the more common and important ones are those based on specific spatial boundaries. Combined with the hybrid spatial index model in Section 3.2 of this paper, the spatial query of the point cloud is divided into two phases, i.e., Hilbert tree filtering phase and R*-tree filtering phase. Figure 9 illustrates the general framework of point cloud spatial range query processing. The Hilbert tree filtering phase uses a relatively low computational cost to find a set of partition encoding candidate sets that intersect with the spatial query range; the R*-tree filtering phase reads the information of the corresponding spatial partition's R*-tree nodes from MongoDB based on the mesh encoding candidate sets and deserializes them into memory for precise querying to obtain the final query results.

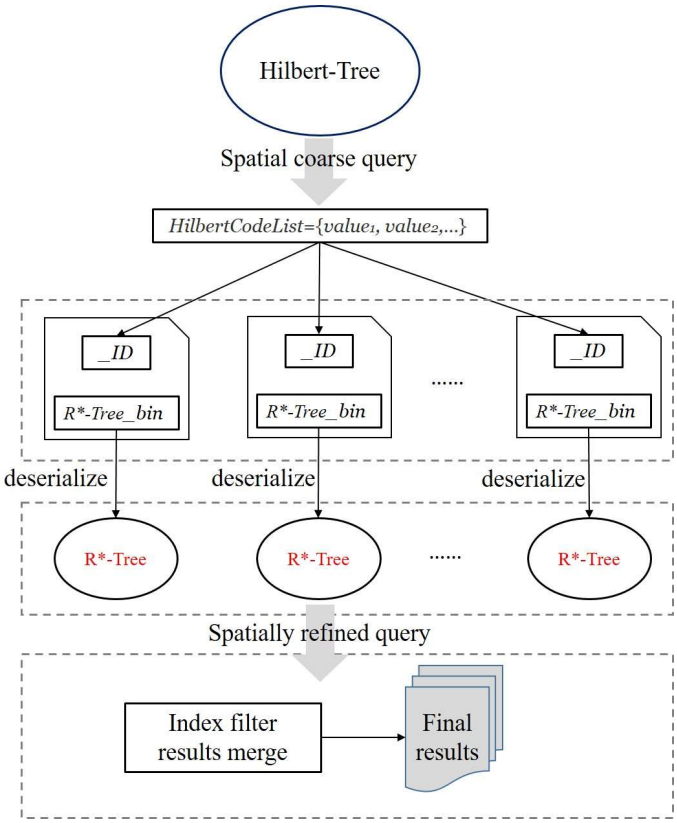

Figure 9. The general framework of spatial range query.

The algorithm for performing point cloud spatial queries is described below.
Algorithm Input: spatial query boundary;
Algorithm Output: point cloud collection;

- Step 1: Obtain the given query boundary information;
- Step 2: Compute a Hilbert grid code set from the Hilbert tree based on the given spatial query boundary to implement a spatial coarse query to clip the query null;
- Step 3: Traverse each Hilbert code in the grid code set to obtain information about the corresponding R*-tree node represented as a binary block in MongoDB;
- Step 4: Deserialize the R*-tree node information represented as binary blocks into memory, retrieve leaf nodes from R*-tree that satisfy the query conditions according to the given spatial query boundaries, and obtain point cloud data from leaf nodes that satisfy the conditions;
- Step 5: Merge and return all R*-tree filtering results and finish the query.

4. Performance Evaluation

4.1. Data Description and Experimental Platform

The point cloud dataset provided by Astyx was selected for this study. This dataset was acquired using a Velodyne VLP-16 sensor (10 Hz, 16 laser beams, 100 m range) installed in the vehicle. The dataset contains a high density of ground points as well as other features such as vegetation with attributes such as 3D coordinates (x, y, z), laser ID, reflectivity, and timestamps, for a total of approximately 11 million point cloud data points. The point cloud index construction and spatial query algorithms were implemented using Java17 with Oracle version 11.2.0.1.0 and MongoDB version 4.4.0, which were run on the Windows 10 64-bit operating system configured with an Intel® Core™ i7-9750H CPU @ 2.60 GHz and 16 GB RAM.

4.2. Performance Analysis

To validate the correctness and effectiveness of the hierarchical index structure and storage scheme for mobile laser point cloud data proposed in this paper, the effectiveness of the Hilbert tree optimization was validated, the indexing performance was evaluated, and the comprehensive test of point cloud spatial querying based on RDBMS and NoSQL was verified using the point cloud dataset described in Section 4.1.

4.2.1. Validation of the Effectiveness of the Hilbert Tree Optimization

To solve the indexing and partitioning problem of large-scale discrete point elements, this paper utilizes Hilbert curves with adaptive space partitioning to construct a Hilbert tree to alleviate data skewing and avoid the complex computational problem of constructing high-order curves. Meanwhile, the Hilbert tree is optimized, and a compact Hilbert tree is proposed. To verify the feasibility and effectiveness of the compact Hilbert tree proposed in this study, we conducted experiments using point cloud data with different scale sizes (percentage of total data amount) to compare the construction time and number of partitions of the Hilbert tree before and after optimization. In this case, the data have been preloaded into the memory and the initial partitioning has been completed. The initial partition order is 3, the maximum partition order is 10, and the space-splitting threshold is 20,000. The specific results are shown in Figure 10.

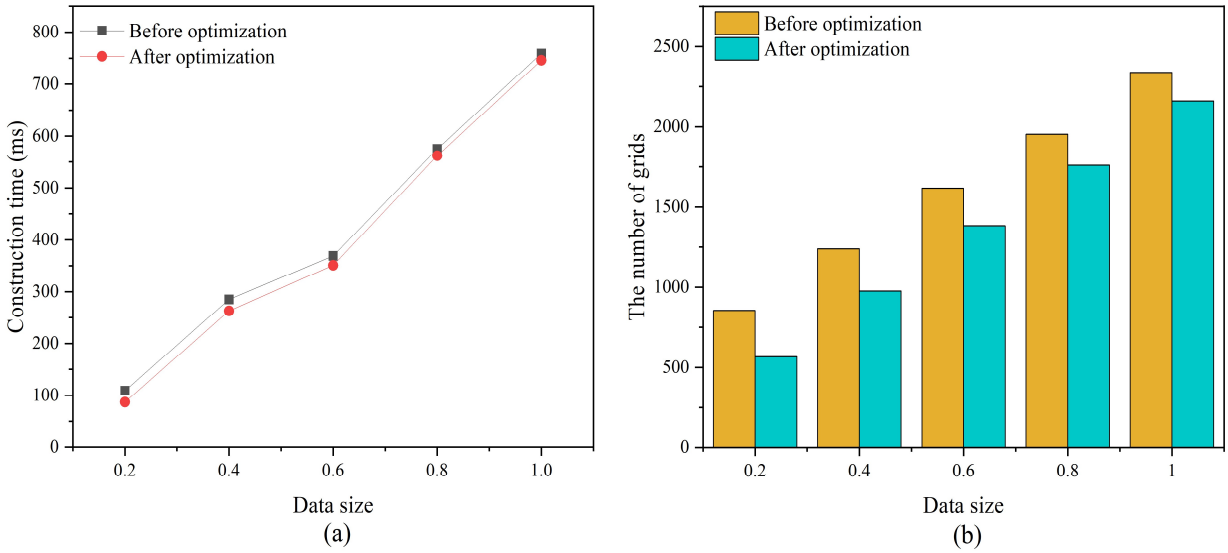

Figure 10. Hilbert tree construction time and the number of partitions before and after optimization. (**a**) Construction time comparison and (**b**) number of partitions comparison.

Figure 10 shows that the optimized Hilbert tree has more excellent construction efficiency and avoids redundant space without data. When the redundant space without

data is eliminated, the number of partitions that must be encoded and calculated is reduced, which narrows the construction time of the Hilbert tree to a certain extent. In addition, the internal structure complexity of the optimized Hilbert tree is reduced, and the space utilization is improved.

For data load balancing, we also examined the situation. The discretization of the data in each partition was investigated with the following standard deviation formula:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{k=1}^{n}(P_i - \mu)^2} \tag{2}$$

where n is the number of grid spaces containing 3D point objects after the end of the division; P_i indicates the number of 3D point objects contained in each grid space; and μ is the average number of 3D point objects contained in each grid space. σ can reflect the discrete situation of the data contained in each grid space. If σ is larger, it indicates that there is a large difference in the number of 3D point objects in each grid space, and there is a more serious skewed data; on the contrary, it has a better realization of load balancing.

We compare the data load balancing between the two cases of uniform division and adaptive division based on 3D space. The order of uniform division is 4; the minimum order of adaptive division is 3, and the maximum order is 10. The experimental results are shown in Figure 11. The adaptive spatial division result corresponds to a smaller σ value, and the discrete degree of the number of 3D point objects contained in each grid is lower. This indicates that adaptive spatial division can alleviate the data skew to some extent and balance the amount of data in each grid space.

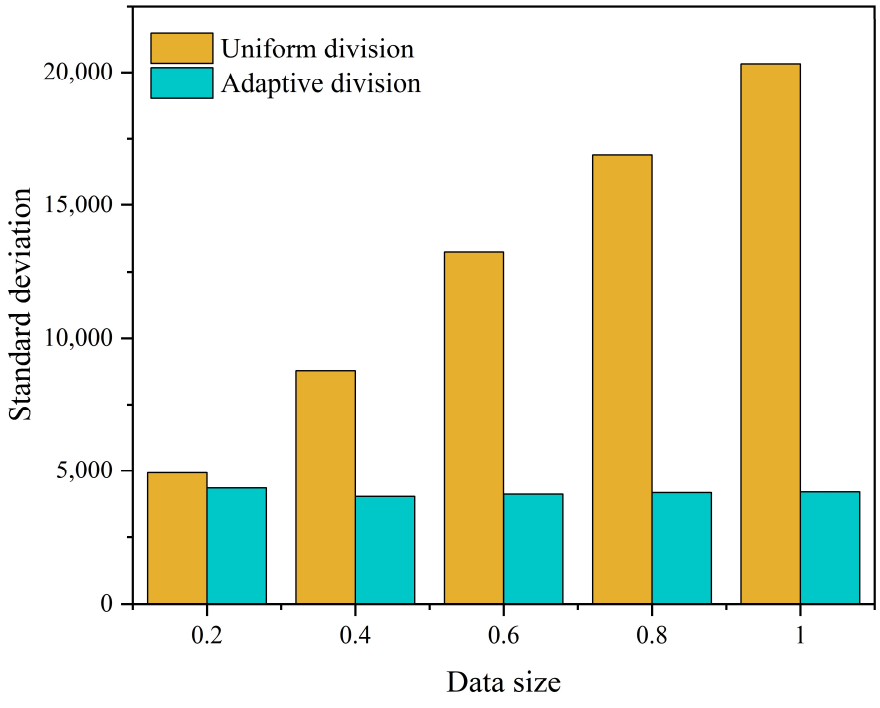

Figure 11. Data skew examination.

4.2.2. Index Performance Evaluation

The node splitting threshold of R*-tree, as a data structure for actually storing point cloud data, affects the maintenance and querying of hierarchical index trees. To examine the effect of the R*-tree node splitting threshold on the hierarchical index tree, three different sets of threshold ranges of (4, 8), (8, 16), and (16, 32) were selected for index construction and spatial query testing. Among them, different sizes of data (percentage of total data amount) were selected to complete the index construction; the spatial query range was set

to 20%, 40%, 60%, 80%, and 100% of the total data volume, respectively; the spatial-splitting threshold of the Hilbert tree was 20,000; and the minimum and maximum coding lengths were 3 and 10, respectively.

Figure 12a shows that a larger R*-tree node splitting threshold significantly increases the construction time of the hierarchical tree. This is mainly because a larger splitting threshold means that each node needs to store more data, and when node splitting and reorganization are performed during the construction of the R*-tree, each node needs to process more child nodes and data entries to complete the optimization and balancing of the tree structure, thus increasing the construction time. In terms of query efficiency, Figure 12b shows that a smaller R*-tree node splitting threshold is not conducive to improving the query efficiency of a hierarchical index tree. When the R*-tree node splitting threshold is increased, each node can accommodate more child nodes, which reduces the length of the search path and allows the query to locate the region where the target data are located more quickly. Therefore, we conclude that the R*-tree node splitting threshold should not be too large or too small to achieve efficient construction and querying of hierarchical trees simultaneously. The suggested node splitting threshold for R*-tree is (8, 16). It is important to note that this choice is not the global optimal solution, and appropriate adjustments must be made when the study area and data size change.

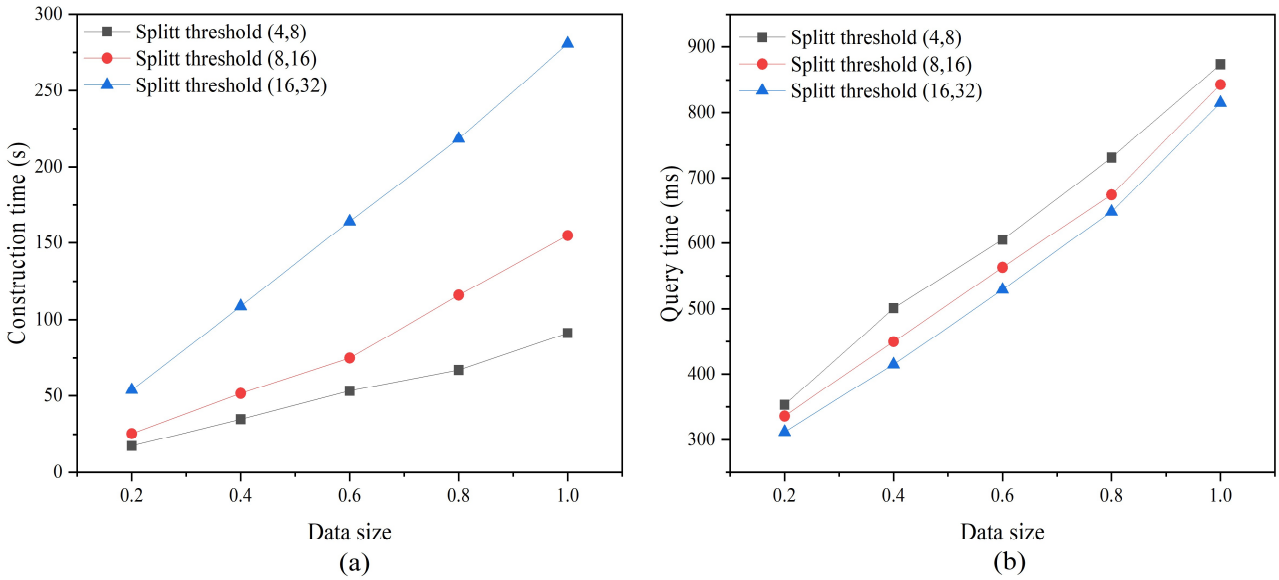

Figure 12. Comparison of index construction and query execution under different R*-tree node splitting thresholds. (**a**) Construction time comparison and (**b**) query time comparison.

Moreover, we selected the two index structures R*-tree and 3DOR-tree [31] with hierarchical index trees for index construction and query performance testing. The selection of the dataset and the setting of the query range were the same as those used in the previous experiment. Among them, the node splitting thresholds of R*-tree and R-tree in the 3DOR-tree were both (8, 16); the node splitting thresholds and depths of Octree in the 3DOR-tree and Hilbert tree in the hierarchical indexing tree were both 20,000 and 10, respectively; and the initial division order of the Hilbert tree was 3. The experimental results are shown in Figure 13.

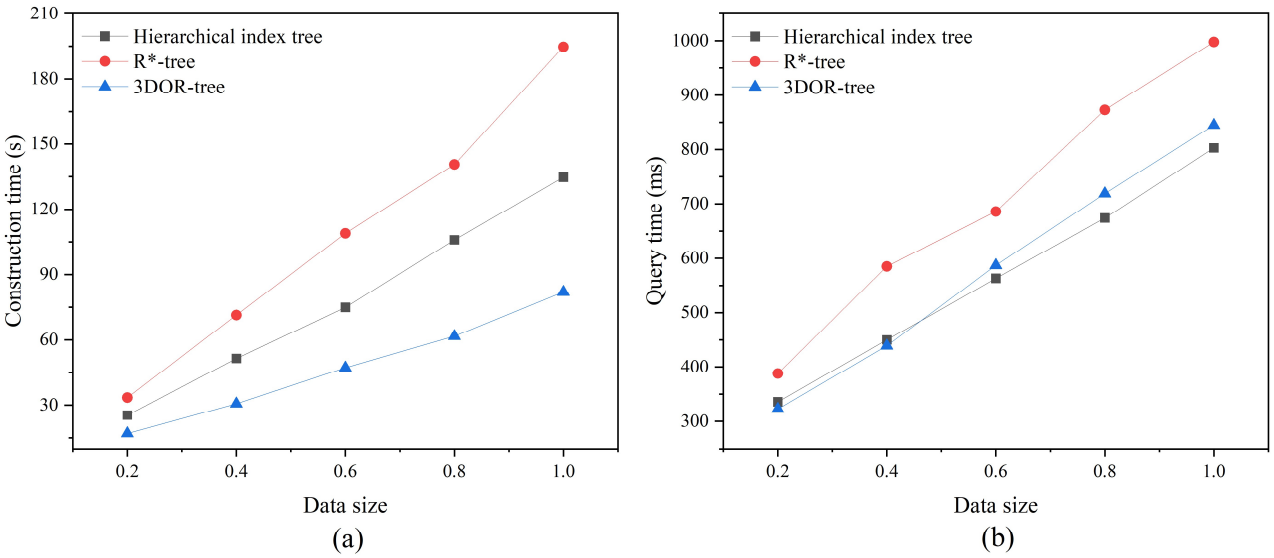

Figure 13. Comparison of construction and query execution of different index structures. (**a**) Construction time comparison and (**b**) query time comparison.

Figure 13 shows that the R*-tree undoubtedly consumes more time in index construction and spatial queries. This is mainly because a single index structure, when facing a large number of spatial objects, has an internal structure that continues to deteriorate with the continuous insertion of data, thus requiring a large amount of time for the maintenance and updating of the index structure. Moreover, the query of the point cloud usually focuses on a specific spatial region, and a large number of redundant and overlapping MBBs may occur in a specific region, resulting in the need to traverse multiple search paths to obtain data that satisfy the query conditions.

For the hierarchical index tree and 3DOR-tree, both had higher index construction and query efficiency and reduced the complexity of index construction through a two-level nested index structure. Moreover, they pruned the query space to provide efficient data query support. In terms of index construction, the construction of a hierarchical index tree takes more time, mainly because the construction of a Hilbert tree is slightly more complex than that of an Octree, in which the space of child nodes must be encoded and calculated, while the index structure is optimized. Secondly, the second level index selection of the hierarchical index tree and 3DOR tree are R* tree and R tree, respectively, and the maintenance and update of R* tree is more time-consuming than R tree. In terms of spatial query, due to the optimization of the Hilbert tree, the space of nodes and search paths that need to be re-solved are reduced and the query efficiency is improved. The reconstructed nodes can more accurately filter the partitioned R* trees that satisfy the query conditions, thus reducing the size of subqueries and further improving the query efficiency. In addition, the query efficiency of R*-tree is higher than that of R tree, and with the increase in query data volume, the advantage of R*-tree is more obviously demonstrated.

4.2.3. Point Cloud Query Efficiency

To investigate the influence of spatial adaptive splitting threshold on data import and query, this experiment uses a complete point cloud dataset for testing, with a minimum coding length of 3 and a maximum of 10, and a splitting threshold of 20,000–30,000, with a spacing of 2000, and the results are shown in Figure 14.

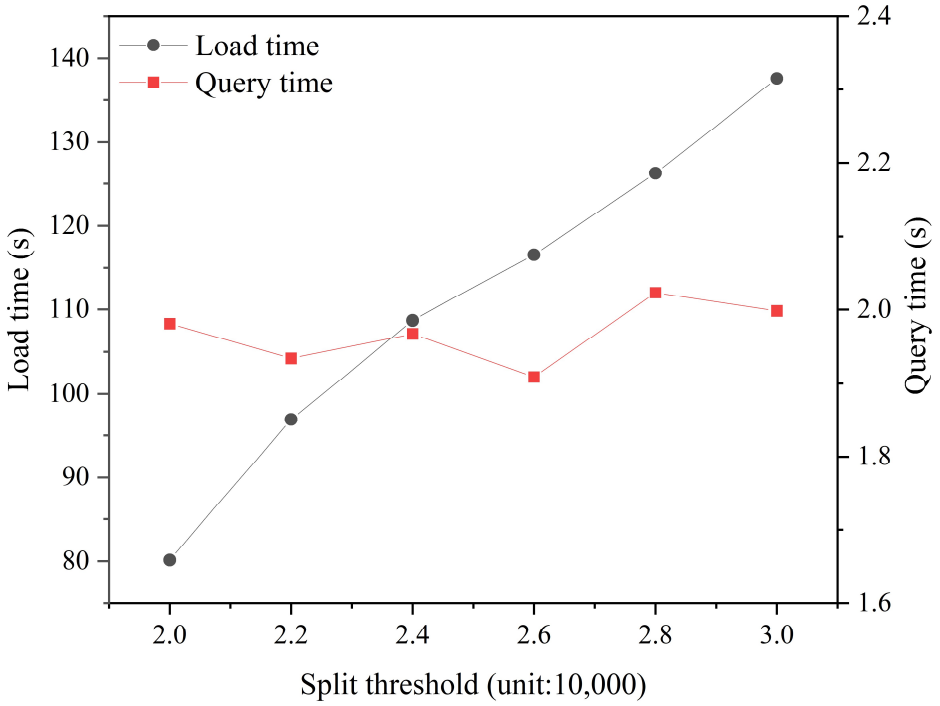

Figure 14. The effect of spatial-splitting thresholds on load and query.

From Figure 14, it is observed that the lower the threshold of spatial adaptive splitting, the shorter the data insertion time, mainly due to the fact that the 3D space undergoes an increased number of splits, which results in the growth of the number of R*-trees, but the size of a single R*-tree decreases, and the time to serialize it into binary blocks is shortened. Meanwhile, observe also that the threshold of spatial adaptive splitting has little effect on the query elapsed time. The reason for this is that, when the sample data are constant, the query time is mainly determined by the size of the query result set.

To verify the efficiency of the point cloud spatial indexing method in this paper for querying in a database management system, point cloud-based spatial extent query was examined. This querying process attempts to identify point clouds that include certain activities within predefined spatial boundaries. The spatial query ranges were set to 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100% of the total data amount. The spatial adaptive division threshold was 25,000, which was chosen based on the experimental results shown in Figure 15. In addition, the node splitting threshold of the R*-tree was (8, 16), and the minimum and maximum coding lengths were 3 and 10, respectively. Efficiency metrics for queries were determined by testing the performance of indexed queries at different spatial dimensional scales, and execution runtimes were analyzed based on the activity of acquiring different amounts of point cloud data and compared with two available point cloud indexing strategies. One of them is based on the SDO_PC extension of Oracle spatial to manage the point cloud, which implements the indexing of the point cloud using R-trees [48]. The SDO_PC is a special data type for point cloud data organization and management, which extends the functions of storage object creation, querying, visualization, etc., for the characteristics of point cloud data. The other is based on encoding 3D point cloud with a Hilbert curve and organizing it into MongoDB. At the same time, B^+-tree indexes are constructed on the encoded fields. To conduct a fair test and ensure the persuasiveness of the results, the three database systems were not optimized, and all the experiments were conducted with the default system settings. The query results are shown in Figure 15.

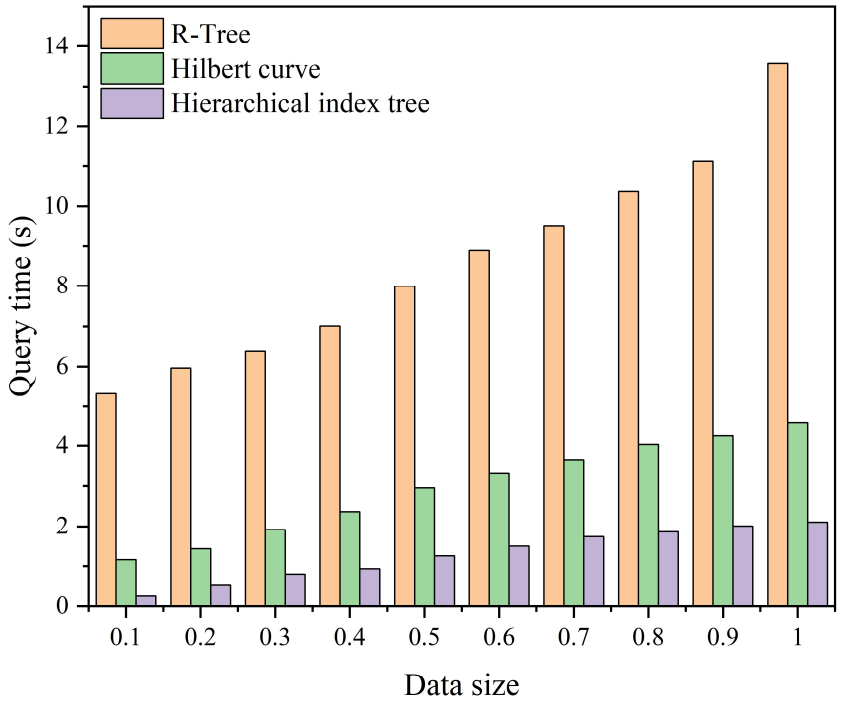

Figure 15. Comparison of point cloud query efficiency supported with different methods.

Figure 15 illustrates the performance of spatial thresholding-based queries. During the point cloud query processing, the Hilbert tree is traversed first to prune the sub-partitions in each partition that exceed the given spatial range and reduce the number of subqueries. From the effectiveness test results, the performance of the hierarchical index structure is much better. As described in Section 3.2 of this paper for the hierarchical index structure, the subqueries utilize the smaller R*-tree instead of the entire index to prune the search efficiently. The proposed index exhibits more efficient spatial query processing compared to the other two point cloud indexing methods.

Figure 15 shows the features and advantages of this paper's method in point cloud spatial queries. The running time of the query increases as the spatial range and data size increase because it needs to access a large number of partitions storing the R*-tree to obtain all the points that satisfy the query conditions. However, it always maintains a better spatial querying performance and shows good scalability. Additionally, when the user accesses a point cloud with a small range, the search prunes it according to the spatial boundaries, thus accessing a reduced set of R*-trees. Moreover, this study constructs a hierarchical index of point clouds using flexible spatial partitioning with R*-tree to provide efficient point cloud query support. The hierarchical index is especially good at handling large-scale spatial queries of point clouds, and increases in the query range have less impact on the query efficiency of the hierarchical index and a greater impact on the other two indexes. The R-tree index retrieves the point cloud data that meets the query conditions through the top-down approach, and with the expansion of the query boundary, multiple search paths must be traversed to obtain the three-dimensional point objects, and the query efficiency is significantly reduced. When using Hilbert curves for point cloud organization, the use of 1D queries instead of 3D queries improves the query efficiency to some extent as the query boundary expands. However, due to the limited curve accuracy, a large number of 3D point objects will be inaccurately included in the query boundary in the index-filtered results. Therefore, after the first filtering of the main index, a large number of records still need to be filtered to accurately obtain the point cloud data, which increases the time and space overhead of executing a complete point cloud spatial query process. Hierarchical indexing reduces the number of 3D queries by clipping the query task through global and local

two-level index filtering and is able to accurately filter the point cloud data that satisfies the query conditions directly from the R*-tree, which makes the query efficiency even better.

5. Conclusions and Discussion

With the goal of efficiently processing large point cloud datasets containing spatial information, this study explores how to utilize NoSQL to provide efficient point cloud data indexing, storage, and query support and proposes a point cloud management method based on the hierarchical spatial indexing model and the MongoDB fusion design storage structure. The experimental results verify the effectiveness of the method proposed in this paper: (1) the Hilbert tree constructed with spatial adaptive partitioning can effectively reduce the computational complexity while avoiding redundant space, thereby alleviating data skewing and fully exploiting the performance of the R*-tree to provide efficient and accurate support for data querying, and (2) when performing spatial range queries on point cloud data, based on the hierarchical indexing architecture, MongoDB outperforms the other three mainstream point cloud management methods. The method first realizes point cloud block management and global index construction through adaptive spatial partitioning and Hilbert curves. Subsequently, local spatial indexing is realized by using the R*-tree. Finally, the high performance and high scalability of MongoDB are utilized to complete the storage structure. As the data volume increases, the features and advantages of the proposed method become increasingly evident. Compared with existing point cloud data storage and query schemes, the spatial index model and storage strategy presented in this paper can effectively satisfy the point cloud data storage requirements of data-intensive spatial applications.

The proposed methodology has a few limitations. In the future, the following research will be conducted, with a focus on the storage and distribution of data:

- Larger point cloud datasets will be selected for performance testing, while the coding accuracy of Hilbert curves will be extended beyond 64 bits to cover larger spatial regions and improve coordinate accuracy;
- Additional types of point cloud data will be selected for testing, such as airborne LiDAR data and fixed LiDAR data, to discuss the wide applicability of the method;
- The method in this study provides a limited variety of point cloud queries and only considers range queries that are widely used in practical engineering. Therefore, we will provide additional point cloud query algorithms, such as *k*NN query, to perform similar queries in the future;
- The methodology proposed in this paper will be used in real point cloud data application scenarios, while a more comprehensive methodology comparison will be carried out to compare the performance of the proposed method with that of other database management systems, e.g., Cassandra and PostgreSQL, to refine the methodology of this paper and apply it in a clustered environment;
- Cloud computing and virtualization technologies provide on-demand, scalable computing resources that have been widely used to support a variety of geospatial studies. Thus, the feasibility of the proposed approach will be explored in other cloud computing environments.

Author Contributions: Yuqi Yang conceived, designed, and performed the experiments and wrote the manuscript; Xiaoqing Zuo and Kang Zhao supervised this study; and Yongfa Li offered helpful suggestions and reviewed the manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (No. 42161067), Yunnan Province Technical Innovation Talent Development Projects (No. 202405AD350058), and Major Science and Technology Projects of Yunnan Province (No. 202202AD080010).

Data Availability Statement: The data that support the findings of this study are openly available at https://github.com/under-the-radar/radar_dataset_astyx (accessed on 24 February 2021).

Acknowledgments: We would like to thank the reviewers for their in-depth suggestions and corrections that helped improve the quality of this paper.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Wang, C.; Hu, F.; Sha, D.; Han, X. Efficient LiDAR point cloud data managing and processing in a hadoop-based distributed framework. *ISPRS Ann. Photogramm. Remote Sens. Spat. Inf. Sci.* **2017**, *4*, 121–124. [CrossRef]
2. Che, E.; Jung, J.; Olsen, M.J. Object Recognition, Segmentation, and Classification of Mobile Laser Scanning Point Clouds: A State of the Art Review. *Sensors* **2019**, *19*, 810. [CrossRef] [PubMed]
3. Poux, F. The Smart Point Cloud: Structuring 3D Intelligent Point Data. Ph.D. Thesis, Université de Liège, Liège, Belgium, 2019.
4. Yang, B.; Haala, N.; Dong, Z. Progress and Perspectives of Point Cloud Intelligence. *Geo-Spat. Inf. Sci.* **2023**, *26*, 189–205. [CrossRef]
5. Vo, A.V.; Hewage, C.N.L.; Russo, G.; Chauhan, N.; Laefer, D.F.; Bertolotto, M.; Le-Khac, N.-A.; Oftendinger, U. Efficient LiDAR Point Cloud Data Encoding for Scalable Data Management within the Hadoop Eco-System. In Proceedings of the 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 9–12 December 2019; pp. 5644–5653.
6. Vo, A.V.; Laefer, D.F.; Trifkovic, M.; Hewage, C.N.L.; Bertolotto, M.; Le-Khac, N.A.; Ofterdinger, U. A highly scalable data management system for point cloud and full waveform lidar data. *Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci.* **2020**, *XLIII-B4-2020*, 507–512. [CrossRef]
7. Vo, A.-V.; Konda, N.; Chauhan, N.; Aljumaily, H.; Laefer, D.F. Lessons Learned with Laser Scanning Point Cloud Management in Hadoop HBase. In *Proceedings of the Advanced Computing Strategies for Engineering*; Smith, I.F.C., Domer, B., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 231–253.
8. Béjar-Martos, J.A.; Rueda-Ruiz, A.J.; Ogayar-Anguita, C.J.; Segura-Sánchez, R.J.; López-Ruiz, A. Strategies for the Storage of Large LiDAR Datasets—A Performance Comparison. *Remote Sens.* **2022**, *14*, 2623. [CrossRef]
9. Ogayar-Anguita, C.J.; López-Ruiz, A.; Rueda-Ruiz, A.J.; Segura-Sánchez, R.J. Nested Spatial Data Structures for Optimal Indexing of LiDAR Data. *ISPRS J. Photogramm. Remote Sens.* **2023**, *195*, 287–297. [CrossRef]
10. Schütz, M.; Ohrhallinger, S.; Wimmer, M. Fast Out-of-Core Octree Generation for Massive Point Clouds. *Comput. Graph. Forum* **2020**, *39*, 155–167. [CrossRef]
11. Wang, W.; Hu, Q. The Method of Cloudizing Storing Unstructured LiDAR Point Cloud Data by MongoDB. In Proceedings of the 2014 22nd International Conference on Geoinformatics, Kaohsiung, Taiwan, 25–27 June 2014; pp. 1–5.
12. Hu, F.; Yang, C.; Jiang, Y.; Li, Y.; Song, W.; Duffy, D.Q.; Schnase, J.L.; Lee, T. A Hierarchical Indexing Strategy for Optimizing Apache Spark with HDFS to Efficiently Query Big Geospatial Raster Data. *Int. J. Digit. Earth* **2020**, *13*, 410–428. [CrossRef]
13. Hanusniak, V.; Svalec, M.; Branicky, J.; Takac, L.; Zabovsky, M. Exploitation of Hadoop Framework for Point Cloud Geographic Data Storage System. In Proceedings of the 2015 Fifth International Conference on Digital Information Processing and Communications (ICDIPC), Sierre, Switzerland, 7–9 October 2015; pp. 197–200.
14. Li, Z.; Yang, C.; Liu, K.; Hu, F.; Jin, B. Automatic Scaling Hadoop in the Cloud for Efficient Process of Big Geospatial Data. *ISPRS Int. J. Geo-Inf.* **2016**, *5*, 173. [CrossRef]
15. Li, Z.; Hodgson, M.E.; Li, W. A General-Purpose Framework for Parallel Processing of Large-Scale LiDAR Data. *Int. J. Digit. Earth* **2018**, *11*, 26–47. [CrossRef]
16. Boehm, J.; Liu, K. NOSQL For Storage and Retrieval of Large LiDAR Data Collections. *ISPRS Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci.* **2015**, *40*, 577–582. [CrossRef]
17. Rueda-Ruiz, A.J.; Ogáyar-Anguita, C.J.; Segura-Sánchez, R.J.; Béjar-Martos, J.A.; Delgado-Garcia, J. SPSLiDAR: Towards a Multi-Purpose Repository for Large Scale LiDAR Datasets. *Int. J. Geogr. Inf. Sci.* **2022**, *36*, 992–1011. [CrossRef]
18. Lokugam Hewage, C.N.; Laefer, D.F.; Vo, A.-V.; Le-Khac, N.-A.; Bertolotto, M. Scalability and Performance of LiDAR Point Cloud Data Management Systems: A State-of-the-Art Review. *Remote Sens.* **2022**, *14*, 5277. [CrossRef]
19. Lu, B.; Wang, Q.; Li, A. Massive Point Cloud Space Management Method Based on Octree-like Encoding. *Arab. J. Sci. Eng.* **2019**, *44*, 9397–9411. [CrossRef]
20. Kim, T.; Lee, J.; Kim, K.-S.; Matono, A.; Li, K.-J. Utilizing Extended Geocodes for Handling Massive Three-Dimensional Point Cloud Data. *World Wide Web* **2021**, *24*, 1321–1344. [CrossRef]
21. Wang, J.; Shan, J. Space-Filling Curve Based Point Clouds Index. In Proceedings of the 8th International Conference on GeoComputation, Kraków, Poland, 23–25 June 2008.
22. Guan, X.; Van Oosterom, P.; Cheng, B. A Parallel N-Dimensional Space-Filling Curve Library and Its Application in Massive Point Cloud Management. *ISPRS Int. J. Geo-Inf.* **2018**, *7*, 327. [CrossRef]
23. Chen, J.; Yu, L.; Wang, W. Hilbert Space Filling Curve Based Scan-Order for Point Cloud Attribute Compression. *IEEE Trans. Image Process.* **2022**, *31*, 4609–4621. [CrossRef] [PubMed]
24. Chen, W.; Zhu, X.; Chen, G.; Yu, B. Efficient Point Cloud Analysis Using Hilbert Curve. In *Proceedings of the Computer Vision—ECCV 2022*; Avidan, S., Brostow, G., Cissé, M., Farinella, G.M., Hassner, T., Eds.; Springer Nature Switzerland: Cham, Switzerland, 2022; pp. 730–747.

25. Elseberg, J.; Borrmann, D.; Nüchter, A. One Billion Points in the Cloud—An Octree for Efficient Processing of 3D Laser Scans. *ISPRS J. Photogramm. Remote Sens.* **2013**, *76*, 76–88. [CrossRef]
26. Tian, S.; Li, X.; Zeng, J.; Wei, Z. The Organization of Point Cloud Data Based on the Compact Octree Model. *J. Phys. Conf. Ser.* **2019**, *1302*, 022047. [CrossRef]
27. Huang, H. Construction of Multi-Resolution Spatial Data Organization for Ultralarge-Scale 3D Laser Point Cloud. *Sens. Mater.* **2023**, *35*, 87. [CrossRef]
28. Zhang, R.; Li, G.; Wang, L.; Li, M.; Zhou, Y. A New Method of Hybrid Index for Mobile LiDAR Point Cloud Data. *Geomat. Inf. Sci. Wuhan Univ.* **2018**, *43*, 993–999.
29. Wang, Y.; Lv, H.; Ma, Y. Geological Tetrahedral Model-Oriented Hybrid Spatial Indexing Structure Based on Octree and 3D R*-tree. *Arab. J. Geosci.* **2020**, *13*, 728. [CrossRef]
30. Zhu, Q.; Gong, J.; Zhang, Y. An Efficient 3D R-Tree Spatial Index Method for Virtual Geographic Environments. *ISPRS J. Photogramm. Remote Sens.* **2007**, *62*, 217–224. [CrossRef]
31. Gong, J.; Zhu, Q.; Zhong, R.; Zhang, Y.; Xie, X. An Efficient Point Cloud Management Method Based on a 3D R-Tree. *Photogramm. Eng. Remote Sens.* **2012**, *78*, 373–381. [CrossRef]
32. Wang, Y.; Yang, L.; Liao, L.; Pan, H. Integrated laser point cloud data storage structure based on octree and 3D R*-tree. *J. Geo-Inf. Sci.* **2017**, *19*, 587–594.
33. Yu, A.; Mei, W. Efficient Management Method for Massive Point Cloud Data of Metro Tunnel Based on R-tree and Grid. *Geomat. Inf. Sci. Wuhan Univ.* **2019**, *44*, 1553–1559.
34. Deibe, D.; Amor, M.; Doallo, R. Big Data Storage Technologies: A Case Study for Web-Based LiDAR Visualization. In Proceedings of the 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 10–13 December 2018; pp. 3831–3840.
35. Pajić, V.; Govedarica, M.; Amović, M. Model of Point Cloud Data Management System in Big Data Paradigm. *ISPRS Int. J. Geo-Inf.* **2018**, *7*, 265. [CrossRef]
36. Deibe, D.; Amor, M.; Doallo, R. Big Data Geospatial Processing for Massive Aerial LiDAR Datasets. *Remote Sens.* **2020**, *12*, 719. [CrossRef]
37. Yao, X.; Mokbel, M.F.; Alarabi, L.; Eldawy, A.; Yang, J.; Yun, W.; Li, L.; Ye, S.; Zhu, D. Spatial Coding-Based Approach for Partitioning Big Spatial Data in Hadoop. *Comput. Geosci.* **2017**, *106*, 60–67. [CrossRef]
38. Di Stefano, F.; Chiappini, S.; Gorreja, A.; Balestra, M.; Pierdicca, R. Mobile 3D Scan LiDAR: A Literature Review. *Geomat. Nat. Hazards Risk* **2021**, *12*, 2387–2429. [CrossRef]
39. Yiğit, A.Y.; Gamze Hamal, S.N.; Ulvi, A.; Yakar, M. Comparative Analysis of Mobile Laser Scanning and Terrestrial Laser Scanning for the Indoor Mapping. *Build. Res. Inf.* **2024**, *52*, 402–417. [CrossRef]
40. Cao, B.; Feng, H.; Liang, J.; Li, X. Hilbert Curve and Cassandra Based Indexing and Storing Approach for Large-Scale Spatiotemporal Data. *Geomat. Inf. Sci. Wuhan Univ.* **2021**, *46*, 620–629.
41. Eldawy, A.; Alarabi, L.; Mokbel, M.F. Spatial Partitioning Techniques in SpatialHadoop. *Proc. VLDB Endow.* **2015**, *8*, 1602–1605. [CrossRef]
42. Kang, Y.; Gui, Z.; Ding, J.; Wu, J.; Wu, H. Parallel Ripley's K function based on Hilbert spatial partitioning and Geohash indexing. *J. Geo-Inf. Sci.* **2022**, *24*, 74–86.
43. Yao, X.; Yang, J.; Li, L.; Ye, S.; Yun, W.; Zhu, D. Parallel Algorithm for Partitioning Massive Spatial Vector Data in Cloud Environment. *Geomat. Inf. Sci. Wuhan Univ.* **2018**, *43*, 1092–1097.
44. Moten, D. Hilbert-Curve. Available online: https://github.com/davidmoten/hilbert-curve (accessed on 23 February 2017).
45. Wang, H.; Belhassena, A. Parallel Trajectory Search Based on Distributed Index. *Inf. Sci.* **2017**, *388–389*, 62–83. [CrossRef]
46. Beckmann, N.; Kriegel, H.-P.; Schneider, R.; Seeger, B. 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, Atlantic City, NJ, USA, 23–25 May 1990; pp. 322–331.
47. Moten, D. Rtree. Available online: https://github.com/davidmoten/rtree (accessed on 1 September 2014).
48. van Oosterom, P.; Martinez-Rubi, O.; Ivanova, M.; Horhammer, M.; Geringer, D.; Ravada, S.; Tijssen, T.; Kodde, M.; Gonçalves, R. Massive Point Cloud Data Management: Design, Implementation and Execution of a Point Cloud Benchmark. *Comput. Graph.* **2015**, *49*, 92–125. [CrossRef]