Article

QRB-tree Indexing: Optimized Spatial Index Expanding upon the QR-tree Index

Jieqing Yu ^{1,2}, Yi Wei ^{2,*}, Qi Chu ² and Lixin Wu ³

¹ NASG Key Laboratory of Land Environment and Disaster Monitoring, China University of Mining and Technology, Xuzhou 221116, China; yujieqing@cumt.edu.cn

² School of Environmental Science and Spatial Informatics, China University of Mining and Technology, Xuzhou 221116, China; gischuqi@cumt.edu.cn

³ School of Geosciences and Info-Physics, Central South University, Changsha 410083, China; wulx66@csu.edu.cn

* Correspondence: wei_yi@cumt.edu.cn; Tel.: +86-15329510050

Abstract: Support for region queries is crucial in geographic information systems, which process exact queries through spatial indexing to filter features and subsequently refine the selection. Although the filtering step has been extensively studied, the refinement step has received little attention. This research builds upon the QR-tree index, which decomposes space into hierarchical grids, registers features to the grids, and builds an R-tree for each grid, to develop a new QRB-tree index with two levels of optimization. In the first level, a bucket is introduced in every grid in the QR-tree index to accelerate the loading and search steps of a query region for the grids within the query region. In the second level, the number of candidate features to be eliminated is reduced by limiting the features to those registered to the grids covering the corners of the query region. Subsequently, an approach for determining the maximal grid level, which significantly affects the performance of the QR-tree index, is proposed. Direct comparisons of time costs with the QR-tree index and geohash index show that the QRB-tree index outperforms the other two approaches for rough queries in large query regions and exact queries in all cases.

Keywords: spatial indexing; QRB-tree index; QR-tree index; region query; rough query; exact query

Citation: Yu, J.; Wei, Y.; Chu, Q.; Wu, L. QRB-tree Indexing: Optimized Spatial Index Expanding upon the QR-tree Index. *ISPRS Int. J. Geo-Inf.* **2021**, *10*, 727. <https://doi.org/10.3390/ijgi10110727>

Academic Editors: Gloria Bordogna and Wolfgang Kainz

Received: 1 August 2021

Accepted: 11 October 2021

Published: 27 October 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

A region query operation is often involved in many processes, e.g., spatial analysis, data sharing, and visualization and mapping [1]. Consequently, the efficiency of this operation is crucial to a geographic information system, especially a system such as Digital Earth [2], which manages a vast amount of spatial data. A region query operation is identical to an intersect operation [3] that determines the features whose boundaries are contained by, overlapping, or touched by a region, e.g., a rectangle, and is often implemented by a spatial index. In real-life applications, the geometry of a feature may be highly complex, and it may be time-consuming to decide whether the feature intersects a query region. A common approach [4] to accelerate the region query when using a spatial index is to use an approximation, usually a minimum bounding rectangle (MBR), instead of the actual geometry for the intersecting test. However, this configuration may yield inaccurate results, in which the retrieved features (referred to as candidate features hereafter) may not exactly intersect with the query region. To obtain an accurate result, an exact test that can examine whether the candidate features intersect with the query region must be performed. The first and second steps of a query are known as filter and refinement steps, respectively. Accordingly, a region query is termed a rough query if it contains only the filter step, and an exact query if it contains both steps [4–6].

The implementation of both steps may be very time-consuming. To accelerate the refinement step, Kothuri et al. [7] introduced an immediate filter to reduce the number

of candidates passed to the refinement step. Features within the query region, which can be determined cost-effectively by considering the relationship between the interior approximation of the query region and MBR of the candidate feature [5], are filtered before the refinement step. In addition, the PostGIS software (<http://postgis.net/> accessed on 13 October 2021) uses a prepared geometry to accelerate the loop of the exact tests in the refinement step.

Although the acceleration of the refinement step is important, the current spatial indices focus mainly on the filter step, possibly because of the considerable scope remaining to optimize the refinement step. In recent decades, numerous spatial indices have been developed, which can be classified as grid-based indices, tree-based indices, or hybrid indices. Grid-based indices, e.g., grid files [8], quad-tree indices [9], Sierpinski curve indices [10], ellipsoidal quad-trees [11], geohashes [12,13], and global multi-scale grid integer coding model (GMGICM) indices [14], are space-oriented indices. These indices divide the entire space into regular or hierarchical grids and assign features to the grids according to their locations and extents. However, it is difficult to determine the size of the grid. The use of either fine or coarse grids may lead to inferior performance. Tree-based indices, e.g., R-trees [15], are object-oriented indices. These indices divide the entire space into regions based on the approximate extent of the target features and later build a balance tree based on these regions. Due to the success of R-trees, many variants have been developed, including the R+-tree [16], R*-tree [17], Hilbert R-tree [18], priority R-tree (PR-tree) [19], R-tree with update memo (RUM-tree) [20], and LAZY R-tree [21]. Singh and Bawa [22] compared the R-tree variants. Notably, in the case of a large number of features, an extremely large tree is generated, which degrades the performance of the approach. Hybrid indices, prepared by combining grid-based and tree-based indices, e.g., the grid-R-tree [23], R-tree with dynamic non-uniform grid and sub R-tree (RGP-tree) [24] and QR-tree [25–27], can overcome the disadvantages of both indices. In a QR-tree, an R-tree is bound to a grid achieved by quad-tree decomposition. As the extent of an R-tree is limited to the extent of the grid to which it is bound, the number of features in the R-tree is reduced, which can reduce the size of the R-tree. By using the QR-tree index, Yang and Hao [28] could realize highly efficient spatial join queries. Following the idea of R-tree variants, many QR-tree variants have been proposed by the research community, including QR+-trees [29] and QR*-trees [30]. Guo et al. [31] combined a QR*-tree with a PMR quad-tree [32] and developed the PQR-tree index. These variants are expected to outperform the QR-tree, similar to the R+-tree, and the R*-tree outperforms the R-tree. However, both the QR-tree and its variants involve several unnecessary tasks that can be avoided in practice; this aspect represents scope for the improvement of these tools. The QR-tree is mainly designed for two-dimensional indexing. However, this tree can be modified or extended to three-dimensional or even four-dimensional indexing. For example, Hohl et al. [33] combined a k-d tree with an octree and designed a hybrid spatiotemporal index. Yang and Huang [34] combined an extended quad-tree index with a three-dimensional (3D) R-tree to manage and visualize massive amounts of point cloud data. Gu et al. [35] combined an octree and a 3D R-tree and proposed a 3D version of the QR-tree index, i.e., the 3DOR-tree index. Wang et al. [36] replaced the R-tree of the 3DOR-tree index with an R*-tree and proposed a 3DOR*-tree index.

Notably, the abovementioned spatial indices are non-distributed indices. Many researchers have attempted to develop a distributing spatial index, e.g., Mouza et al. [37], Malensek et al. [38], Elashry et al. [39], and Xia et al. [40]. Usually, a distributing spatial index originates from a non-distributed spatial index, e.g., the distributing quad-tree [41], distributing R-tree [40,42,43], distributing QR-tree [44], and distributing Hilbert TGS R-tree [45]. As more computational nodes are used, a distributing spatial index is often expected to run faster than its non-distributed version. However, the actual performance of a distributing spatial index is highly related to the operating environment—for example, the number of nodes or the computing capability per node—and involves a performance ceiling for given hardware. As the solution concepts and techniques are similar, the per-

formance of a distributing spatial index is likely influenced by that of the non-distributed version. Therefore, the development of a more efficient non-distributed spatial index can likely help to enhance the efficiency of the distributing spatial index.

In this study, a novel non-distributed spatial index, named the QRB-tree, (where ‘B’ refers to buckets), is developed based on the QR-tree index by simultaneously optimizing the filter and refinement steps of the QR-tree index. Similar to the migration of the QR-tree index to the distributing QR-tree [44], the QRB-tree index can be migrated to a distributing environment in theory. The remainder of the paper is arranged as follows. The second section introduces the ideas and algorithms of the QRB-tree index and methods of optimization. The third section discusses the selection of the maximal level of grids, a key parameter that affects the performance of the QRB-tree index. The fourth section describes several tests conducted to compare the QRB-tree index with the QR-tree index and geohash index. The last section presents the concluding remarks.

2. QRB-tree Index

The general idea of the QR-tree index [25] is shown in Figure 1. In the QR-tree index, space is decomposed into multi-level grids, each of which is assigned a linear code and bound with an R-tree, which is a balance tree in which each leaf node stores the MBR of a feature and each non-leaf node records the MBR of all their children. Consequently, a QR-tree index is composed of many R-tree indices, which fundamentally process the region queries. We refer to the R-tree bound to a grid as the associated R-tree of the grid and the grid as the associated grid of the R-tree. Features that fall completely into a grid, referred to as the subordinate features of the grid, are inserted into the associated R-tree. Hence, in this paper, the grid and R-tree are referred to as the associated grid and associated R-tree of the features, respectively. After the associated R-trees are constructed, they are dumped into files named after the linear codes of the associated grids. For a region, the exact query based on the QR-tree index can be realized through the following steps:

- Grid decomposition: decompose the region into multi-level grids with a quad-tree and compute the linear codes;
- R-tree location: locate the index files of the associated R-trees for the grids from the disk with their linear codes;
- R-tree loading: load the index files from the disk and build memory R-trees;
- R-tree search: search each of the memory R-trees by the region in the memory to retrieve candidate features;
- Feature elimination: eliminate the candidate features that do not intersect with the region.

Figure 1. General concept of the QR-tree index. The red object marks the features to be indexed. The features are first assigned to grids of different levels according to their locations and extents. Next,

an R-tree index is assigned for each grid. Each node of the R-tree records either the MBR of a feature (in the case of a leaf node) or the MBR of all its children (in the case of a non-leaf node). The 'M' inside each node of the R-tree represents the MBR.

The first four steps correspond to the filter step of an exact query, i.e., a rough query, and the last step corresponds to the refinement step of the exact query, which is nearly an exact test.

As only the subordinate features of a grid are inserted into its associated R-tree, the number of subordinate features decreases, and the area of overlap between nodes in the associated R-tree reduces. The former and latter phenomena lead to the reduced height of the R-tree and lower possibility of multipath searching, respectively, both of which accelerate the query. Nevertheless, there remains certain scope to enhance the QR-tree index. According to our tests, the last four steps contribute to the majority of the total time cost. Methods to optimize the third and fourth step are presented in Section 2.1, and that for the last step is described in Section 2.2. The optimized QR-tree index is named the QRB-tree index. Figure 2 shows an overview of the QRB-tree index, in which the light blue parts indicate the optimizations over the QR-tree index. The implementations of the QRB-tree index are described in Sections 2.3 and 2.4

Figure 2. Features are distributed into grids of different levels according to their extents and locations, with each of the grids being attached to an R-tree and a bucket. When a range query is implemented, the corner, border, and interior grids covered by the query range are calculated. Features intersecting with the query range are filtered by loading and searching the relevant R-trees or by loading the relevant buckets.

2.1. Optimizations of the R-tree Loading and Search Steps

Since the volume of a QR-tree index in a geographic information system may be very large, only a portion of the index, which is related to the query region, can be loaded in the memory for searching. The R-tree loading step is usually implemented using a recursive process (see the pseudocode presented in Algorithm 1 or the codes in the Boost library (<https://www.boost.org> accessed on 13 October 2021) or the Nushoin's repository (<https://github.com/nushoin/RTree> accessed on 13 October 2021), in which the child nodes are recursively loaded from the disk into memory. The recursive process results in (a) extensive work for the CPU, as frequent operations of context switching are required, and (b) extensive work regarding I/O access, as the loading of the whole index is divided into many instances of individual I/O access (second line of the pseudocode). One

approach to address the first problem is to employ a stack to convert the recursive process into a non-recursive process (e.g., Libspatialindex <https://github.com/libspatialindex> accessed on 13 October 2021). Another approach is to avoid the use of the R-tree index where possible, as discussed later in this section. This paper adopts both of these strategies to address the first problem. The solution to the second problem is straightforward: we load all the data of the index from the disk into the memory at once and manipulate the bytes in the memory.

Algorithm 1 RLoad ()

BEGIN

1. **IF** the current node is a leaf node:
2. Read the block of the node from the disk
3. **ELSE**:
4. **FOR** each child of the current node:
5. RLoad ()
6. **END FOR**
7. **END IF**

END

After the R-trees are loaded, each tree must be searched to obtain the candidate features in the R-tree searching step. If the query region is extremely large, a vast number of nodes are searched, rendering the procedure time-consuming. In the QR-tree index, the query region is first decomposed into multi-level grids, and the associated R-trees of these grids are located and loaded for further searching. However, several of these grids may be interior grids (an example is shown in Figure 3), whose boundaries are within the query region. Thus, all the subordinate features of the interior grids must be retrieved. However, in the current QR-tree index, the associated R-trees of the interior grids must be searched to obtain actual data (e.g., IDs or pointers of features), as they are recorded only in the leaf nodes of the associated R-trees. To avoid this non-necessary work and reduce the time cost, we introduce a bucket in every grid in the QRB-tree index. The bucket directly stores the actual data of the subordinate features for each grid. We refer to the bucket as the associated bucket of the grid and the grid as the associated grid of the bucket. Once a bucket is installed for each grid, the subordinate features of the interior grids can be accessed directly from their associated bucket rather than their associated R-tree.

Figure 3. Interior grids and overlapping grids in a query region.

Figure 4 illustrates the data structure of the QRB-tree index. The subordinate features of a grid are simultaneously inserted into the associated R-tree and associated bucket of the grid. For a query region, the subordinate features of the interior grids pertaining to the region are retrieved directly from their associated buckets, and those of the overlapping grids are retrieved from their associated R-trees. Unlike retrieval from an associated R-tree, retrieval from an associated bucket involves no search procedure; thus, the search time can be eliminated. In addition, since only the feature pointers are stored in a bucket, the size of an associated bucket is always smaller than that of an associated R-tree under the

same grid, which reduces the loading time. Consequently, the region query based on the QRB-tree index is faster than that of the QR-tree index.

Figure 4. Data structure of the QRB-tree index.

2.2. Optimization of the Feature Elimination Step

During the approximation of a feature by its MBR, several of the retrieved features may not actually intersect with the query region. For an exact query, these features should be eliminated. As argued previously, the retrieved features pertain to either the interior grids or overlapping grids. The features retrieved from interior grids must intersect with the query region and hence need no further examination. Nevertheless, the features retrieved from the overlapping grid may not intersect with the query region and hence need further examination. Overlapping grids can be further divided into corner and border grids. Corner grids are grids that cover the corners of the query region, while border grids are the overlapping grids minus the corner grids. Figure 5 shows an example of the grids covered by a query rectangle. Although a polygon, a rectangle, or a circle can be a query region, a rectangle is the most common case. In the following section, the query region is limited to a rectangle.

Figure 5. Grids covered by a rectangle. The dashed rectangles indicate the features' MBRs.

Owing to the shape of the rectangle, the retrieved features from the border grids must intersect with the rectangle. For example, if the top border of the rectangle is crossed by the MBR of a subordinate feature of a border grid (e.g., feature 'A' in Figure 5), the top border of the rectangle must be crossed by the feature itself. This aspect is guaranteed by the constraints of the MBR and border grid. Under this circumstance, feature 'A' can lie neither to the left nor to the right of the rectangle, as it is limited by the border grid, which lies between the left and right borders of the rectangle. Similarly, the feature can lie neither to the top nor to the bottom of the rectangle, as the MBR, which defines the lowermost and topmost borders of the feature, crosses the rectangle's top border. Consequently, it is not necessary to check the features retrieved from border grids.

The MBRs of the retrieved features from the corner grids may either cover or not cover the corners of the rectangle. Accordingly, we can sort these features into corner and non-corner features. Similar to the subordinate features of the border grids, non-corner features (e.g., features '2', '4', '5', and '8' in Figure 5) do not need to be examined further, as they must intersect with the query rectangle if they are retrieved. However, ambiguity exists for the corner features, as they may either intersect (e.g., features '1' and '6' in Figure 5) or not intersect (e.g., features '3' and '7' in Figure 5) with the rectangle in reality. Therefore, corner features must be examined to identify the features that truly intersect with the rectangle. To exploit this aspect, we must separate the corner and non-corner features, as both of these features are simultaneously retrieved from the associated R-tree of the corner grid. Note that the corner features are identical to the features retrieved by the corners of the query rectangle from the associated R-tree of the corner grid. Therefore, only features that are retrieved by each corner of the query rectangle need to be considered for elimination, which accelerates the feature elimination step.

2.3. Insert Algorithm

An insert algorithm is used to insert a feature into an existing QRB-tree index. For a feature, we determine the associated grid of the feature and insert the feature into the associated R-tree and associated bucket of the grid. After all of the features are inserted, the associated R-trees and associated buckets are dumped into files named after the linear code of the grid for further use.

The determination of the associated grid of a feature is based on certain rules: the feature's MBR does not cross the border of its associated grid, and the level (l) of the associated grid is as large as possible while not exceeding a preset value known as the maximal grid level (L^0). L^0 is introduced to ensure the high performance of the region query. This aspect is discussed in the third section. Equation (1) is used to compute l for the associated grid of a feature.

$$l = \min \left\{ L^0, \left\lfloor \log_2^{\Omega_x} \right\rfloor, \left\lfloor \log_2^{\Omega_y} \right\rfloor \right\} \quad (1)$$

where Ω_x and Ω_y are the dimensions of the feature's MBR and $\lfloor \cdot \rfloor$ indicates a rounding down operation.

Both L^0 and l are relative to a base level. For a grid decomposed based on the extent of the dataset, the base level is zero. However, the base level may be non-zero for a grid decomposed based on a globe. In this case, the base level is defined as the level at which the whole dataset is totally included by a grid unit. Without loss of generality, the base level is excluded from L^0 and l in the following analyses. However, the base level should be considered in practice.

The linear code of an associated grid can be calculated using l and the feature's MBR. The code is based on the index number of the four corners of the MBR along the horizontal and vertical dimensions in the l th level grid. Theoretically, the index numbers of the four corners should be identical, as the MBR is supposed to be fully contained by its associated grid. The Morton coding method [46] can be used to determine the linear code of a grid.

The method of inserting a feature into its associated R-tree is the same as that of inserting a feature into a traditional R-tree. The method of inserting a feature into the associated bucket, which is similar to pushing a new element into an array, is straightforward. The pseudocode for the insert Algorithm 2 can be expressed as follows.

Algorithm 2 Insert

INPUT: ptr , pointer of the feature to be inserted
 MBR , minimum bounding rectangle of the feature to be inserted
 L^0 , maximal grid level
BEGIN
 1. Calculate the level (l) for the associated grid of the feature according to Equation (1);
 2. Determine the associated grid of the feature at the l th level and compute its linear code;
 3. Locate the associated R-tree and associated bucket of the grid by its linear code;
 4. Insert ptr into the associated bucket;
 5. Insert the tuple (ptr, MBR) into the associated R-tree;
END

2.4. Search Algorithm

The search algorithm is used to implement a region query. To perform an exact query, both the filter and refinement steps must be implemented in the QRB-tree index. In many cases, when referring to spatial queries, we refer to the filter step [4], i.e., the rough query. As the features retrieved by rough queries may be adequately accurate for certain applications, such as real-time visualization, we present two algorithms, rough and exact, for rough queries and exact queries, respectively. The rough algorithm implements the filter step, while the exact algorithm implements both steps with several optimizations to attain superior performance.

Given a rectangle for a query, the rectangle is first decomposed into grids at each level between 0 and L^0 . The grids are sorted into interior and overlapping grids if the rough algorithm is adopted or interior, border, and corner grids if the exact algorithm is adopted. Subsequently, the associated R-trees of the overlapping grids or corner grids and border grids are searched within the rectangle by using the algorithm of the traditional R-tree, and the associated buckets of the interior grids are read directly without any further searching. In the exact algorithm, the corners of the rectangle are used to search the associated R-trees of the corner grids to differentiate corner and non-corner features. An exact test is performed for each corner feature. Disjointed features are eliminated from the candidate features.

The decomposition of a rectangle at the i th level can be performed considering the minimal and maximal grid indices of the rectangle on the i th level grids. The sorting of the interior, overlapping, border, and corner grids is straightforward according to their definitions. The pseudocodes for the two Algorithms 3 and 4 are presented.

Algorithm 3 Rough search

INPUT: *rect*, rectangle for query
 L^0 , maximal grid level
OUTPUT:
ftrs, features retrieved by the algorithm

BEGIN

1. **FOR** each level *l* between 0 and L^0
2. Calculate the overlapping and interior grids for the rectangle at the *l*th level;
3. Append the linear codes of the two grids to *ArrOlp* and *ArrItr*;
4. **END FOR**
5. **FOR** each grid *g* in *ArrOlp*
6. Locate and load the associated R-tree and the associated bucket of grid *g*
7. Search the associated R-tree by *rect*, and return the pointers of the candidate features;
8. Append the feature pointers to *ftrs*;
9. **END FOR**
10. **FOR** each grid *g* in *ArrItr*
11. Locate and load the associated bucket of grid *g*;
12. Read the features from the associated bucket and append them to *ftrs*;
13. **END FOR**
14. **RETURN** *ftrs*

END

Algorithm 4 Exact search

INPUT: *rect*, rectangle for query
 L^0 , maximal grid level
OUTPUT:
ftrs, features retrieved by the algorithm

BEGIN

1. **FOR** each level *l* between 0 and L^0
2. Calculate the corner, border and interior grids for the rectangle at the *l*th level;
3. Append the linear codes of the corner grids and rectangle's corners corresponding to the corner grids to *ArrCrn*;
4. Append the linear codes of the border and interior grids to *ArrBrd* and *ArrItr*, respectively;
5. **END FOR**
6. **FOR** each grid *g* and its corresponding corner *c* in *ArrCrn*
7. Locate and load the associated R-tree and associated bucket of grid *g*;
8. Search the R-tree by *rect* and return a set *Bls_rct*;
9. Search the R-tree by the corner *c* and return an array *Bls_crn*;
10. **FOR** each feature *p* in *Bls_crn*
11. **IF** feature *p* does not intersect with *rect*
12. Remove *p* from *Bls_rct*;
13. **END IF**
14. **END FOR**
15. Append all features in *Bls_rct* to *ftrs*;
16. **END FOR**
17. **FOR** each grid *g* in *ArrBrd*
18. Locate and load the associated R-tree and associated bucket of grid *g*;
19. Search the associated R-tree by *rect* and obtain an array *Bls*;
20. Append all features in *Bls* to *ftrs*;
21. **END FOR**
22. **FOR** each grid *g* in *ArrItr*
23. Locate and load the associated bucket of grid *g*;
24. Read the feature pointers from the associated bucket;
25. Append all features to *ftrs*;
26. **END FOR**
27. **RETURN** *ftrs*

END

3. Choice of Maximal Grid Level (L^0)

A feature can be assigned to a grid of any level not larger than L^0 in the QRB-tree index depending on its location and extent. A smaller feature is more likely to be assigned to a lower-level grid (leading to a larger grid). Given a query region, a lower level means fewer interior grids. In this framework, the advantages of the QRB-tree index cannot be exploited (fewer buckets are used). In contrast, a higher level means more grids. In this case, more associated R-trees and associated buckets are involved, which is detrimental to the performance. We experimentally examined the impact of L^0 on the performance and clarified the method to select L^0 .

3.1. Impact of L^0 on the Performance

We randomly generated a dataset with more than 100 million features with different sizes. For simplicity, only the MBR instead of the geometry was generated. Next, six rectangles with increasing extents were simulated to implement region queries through the QRB-tree index. Figure 6a illustrates the extents of the six rectangles, and Figure 6b shows the magnified view of the smallest rectangle in the simulated dataset. For convenience, the R-tree files and bucket files are referred to as auxiliary files. Figure 7 shows the time cost for each individual step against L^0 among these rectangles and the numbers of R-tree files, bucket files, and auxiliary files accessed during these queries. The total time cost first decreases and later increases, indicating a ‘V’ shape, as L^0 increases. The value pertaining to the lowest point of the ‘V’ turn is the optimal choice for L^0 . However, it is difficult to locate such a point in practice, as it may change across cases.

Figure 6. Six query rectangles and the simulated dataset. (a) Six query rectangles. The number near each rectangle is its ID. The outside rectangle denotes the extent of the simulation dataset. (b) Magnified view of rectangle ‘1’ in the simulated dataset.

Figure 7 shows that (a) the time costs of the R-tree loading and R-tree searching decrease as L^0 increases, (b) the time costs of the bucket loading and auxiliary file location increase as L^0 increases, and (c) the contributions of the time cost of the remaining processes are insignificant and can be neglected. These trends can be considered to determine the value for L^0 . In particular, the selected L^0 should not exceed a critical value, beyond which the time costs of the bucket loading or auxiliary file location steps significantly increase. The former and latter critical values are referred to as the bucket threshold (L_b^0) and auxiliary threshold (L_a^0), respectively, and they are discussed in the following subsections.

Figure 7. Time cost (t) of each step in a region query using different rectangles against L^0 for the QRB-tree index and total number (n) of grids covered by each rectangle. ‘R_load’, ‘B_load’, ‘A_locate’, and ‘Others’ refer to the R-tree loading step, bucket loading step, auxiliary file locating step, and remaining steps of a region query, respectively. The subplots from left to right and top to bottom correspond to the query by the six rectangles shown in Figure 6a, in sequence.

3.2. Determination of L_b^0

Given a query region, the total size of the loaded buckets is independent of the selection of L^0 , because the numbers of features associated with these buckets are similar. Figure 8 shows the total and average sizes of the buckets and R-trees that are loaded during the region query by each rectangle. The total size of the loaded buckets increases as L^0 increases but promptly converges. In addition, the average size of the buckets considerably decreases as L^0 increases. A considerable amount of seek time and rotational latency is required to load a chunk of data when it is broken into many pieces and distributed over a discontinuous disk space. Consequently, the time cost of the bucket loading steps increases as L^0 increases, as shown in Figure 7.

Figure 8. Total (Total) and average (Avg) sizes of the buckets (B) and R-trees (R) accessed during the region query by each rectangle. The horizontal dashed line in each subplot indicates the location of 4 KB on the right axis. For certain rectangles, the total size of the buckets is extremely small; thus, these buckets appear to be missing in the figures.

Each file is associated with a minimum amount of space, named the allocation unit size (s^0), to be stored in a disk [47]. For a chunk of data, the number of allocation units required for storage is considerably large, which significantly increases the time cost of loading if the chunk is divided into pieces smaller or considerably smaller than s^0 . Accordingly, the time cost of bucket loading is significantly increased if the selected L^0 is associated with buckets with an average size less than s^0 . In other words, L_b^0 should be associated with buckets with an average size larger than s^0 . Consider a case in which m normally distributed features are available in a dataset, and k bytes are required to store a pointer for a feature in the bucket. The average size of the buckets at $L^0 = l$ can be obtained as

$$s = (mk) / \sum_{i=0}^l 4^i \quad (2)$$

In this simulation, m , k , and s^0 are 10^8 , 8, and 4 KB, respectively. According to Equation (2), s is equal to 2.2 KB and 8.9 KB at $L^0 = 9$ and $L^0 = 8$, respectively. Therefore, L_b^0 should theoretically be 8 in this simulation. As shown in Figure 8, the average size of the buckets decreases to less than s^0 when L^0 increases to 9. Figure 9 plots the number and percentage of the bucket files whose actual sizes are less than 4 KB for this simulation. A sudden increase in the number and percentage of bucket files can be observed at $L^0 = 9$.

In this case (i.e., $L^0 = 9$), the time cost of bucket loading dramatically increases (Figure 7), which confirms the theoretical value for L_b^0 , as discussed previously.

Figure 9. Number (n) and percentage (p) of the bucket files whose actual sizes are less than 4 KB.

3.3. Determination of L_a^0

In the QRB-tree index, each grid is associated with an auxiliary file. The time cost of the auxiliary file location depends on the number of auxiliary files to be located (red line in Figure 7) and the total number of auxiliary files. The former value is related to the extent of the query region, and both values increase exponentially as L^0 increases. Figure 10a shows the normalized time cost for the auxiliary file location in this simulation. Regardless of the extent of the query region, the time cost of the auxiliary file location increases exponentially as L^0 increases. In this context, it is difficult to determine L_a^0 as the time cost of the auxiliary file location grows steadily as L^0 increases. However, the conditions change in terms of the relative contribution. Figure 10b shows the percentage of time costs for the auxiliary file location steps within different region queries as L^0 increases. The percentages increase as L^0 increases. Although the values vary as the extent of the query region increases, the values in nearly all cases increase suddenly when L^0 reaches 7, which suggests that L_a^0 should be 6 in this case. However, this value can be generalized, as it is deduced from the relative contribution, which is independent of the effect of the dataset size and hardware environment.

Figure 10. Time costs associated with auxiliary file location in different region queries as L_a^0 increases. The first and second rectangles are excluded from this figure because their time costs are extremely small (Figure 7) and might easily be affected by the measurement error. (a) Normalized time cost (t_n) obtained by dividing the maximal time cost that appears in each region query; (b) relative time cost determined according to the percentage (p) of the contribution of the auxiliary file location step.

4. Tests and Comparisons

We performed several region queries by using the simulation dataset presented in Section 3 with the first five rectangles shown in Figure 6a to compare the QRB-tree index with the QR-tree index. Since only the MBRs of the features are available in the simulated dataset, rough queries rather than exact queries were performed in this comparison. All queries were implemented on a 64-bit desktop system with a 3.20 GHz CPU, 16 GB memory, and 1 TB mechanical hard disk at $L^0 = 6$ in a non-distributed environment. Figure 11a shows the time cost of the rough query based on the two indices. Although the performances are similar when a small rectangle is encountered, the QRB-tree index outperforms the QR-tree index when a large query rectangle is chosen because more interior grids are present in a larger rectangle than in a smaller rectangle. More interior grids mean that more buckets are used instead of R-trees in the query operations; hence, the operation is faster. Moreover, we simulated four additional datasets with an increasing density of features in the tests. The actual geometry was replaced by the MBR of a feature in each dataset. Figure 11b shows the time costs of the queries on the four datasets when using a uniform query rectangle. The QRB-tree index outperforms the QR-tree index. The improvement seems more significant in a dense dataset than in a sparse dataset because more subordinate features of an interior grid are available in a dense dataset, leading to a larger number of nodes in the associated R-tree of the grid and more time spent on the R-tree loading step with the QR-tree index.

Figure 11. Comparison of the time cost (t) associated with the QR-tree index (t_{QR}) and QRB-tree index (t_{QRB}) for testing on the simulated datasets. The percentage of improvement (p) for the QRB-tree index over the QR-tree index, calculated as $(t_{QR} - t_{QRB})/t_{QR}$, is shown on the right axis. (a) Time costs against the IDs of the first five rectangles shown in Figure 6a. The last rectangle in Figure 6a is excluded because its time cost for the QR-tree index is too large for the others to be clearly shown. (b) Time costs of region queries against the feature density, which is defined by the number of features per unit area.

We conducted several additional tests based on a real dataset provided by the Ordnance Survey of the United Kingdom (<https://osdatahub.os.uk/downloads/open/OpenMapLocal> accessed on 13 October 2021) in the same environment as that for the test simulations, in order to further compare the two indices. The dataset contained approximately 16 million polygons, 6 million lines, and 0.5 million points. According to the theory presented in Section 3, L^0 was selected as 6. Five rectangles numbered 1 to 5 in order of increasing extent were employed in the tests. In each test, both rough and exact queries were performed, and the refinement method for the exact query of Kothuri et al. [7], referred to as the traditional refinement method hereafter, was adopted to perform a comparative analysis with the proposed refinement method. In the method reported by Kothuri et al. [7], features that are fully contained in the query region are screened in advance through the non-intersection test between the interior approximation of the query region and the MBR of the candidate feature. In this study, the interior approximation of the query region was

the query rectangle. The overlap test in the searching algorithm of the associated R-tree performed most of the work for the non-intersection test between the query rectangle and MBR of a feature in the QR-tree index and QRB-tree index. Therefore, we implemented the screening operation of the traditional refinement method in the overlap test procedure to accelerate the process.

Figure 12 shows an overview of the real dataset and five rectangles. For reference, the geohash index [48], a high-performance spatial index, was also implemented in this experiment. Figure 13 shows the performance comparison of the geohash and index based on the PostGIS software, which is an extremely popular spatial database in the community. Moreover, the figure shows the performance of the geohash against the grid level, a key parameter in the geohash index. The geohash outperforms the index based on PostGIS. Nevertheless, the performance of the geohash varies as the grid level changes. To demonstrate the superiority of the proposed method, we set the grid level of the geohash index as 4, at which its performance is the highest, in the following comparisons.

Figure 12. Dataset and rectangles used in the real experiment. The rectangles are numbered 1 to 5.

Figure 13. Comparison of the time cost associated with the geohash index (t_{GH}) and PostGIS software (t_{PG}).

Since geohash uses a globally decomposed grid, a similar grid was used for the other two indices to ensure a fair comparison. The base level for the other two indices was 6. Figure 14a shows the time costs of rough queries based on these indices. The time costs are similar when the query region is small (e.g., rectangles 1 and 2). However, the time costs of the QR-tree index and geohash index increase more rapidly than that of the QRB-tree index as the extent of the query region increases. For large query regions, the QRB-tree index outperforms the other two indices. Figure 14b compares the time costs of the exact

queries based on these indices. The QRB-tree index exhibits the highest performance on exact queries, followed by the QR-tree index and geohash index.

Figure 14. Comparisons among QRB-tree (QRB), QR-tree (QR), and geohash. The right axis indicates the percentage of improvement (p) over geohash, i.e., $(t_{Geohash} - t_X)/t_{Geohash}$, where X refers to QR or QRB. (a) Comparison of rough queries. (b) Comparison of exact queries. The refinement methods used in the exact queries for the three cases are the proposed, traditional, and traditional.

To compare the proposed and traditional refinement methods, we implemented the two refinement methods in the exact queries based on both the QRB-tree index and QR-tree index. Figure 15 shows the results of the comparison. The proposed refinement method always consumed less time than the traditional refinement method.

Figure 15. Comparisons between the proposed (prop) and traditional (tran) refinement methods. The two methods are adopted in the exact queries based on the QR-tree index (QR) and QRB-tree index (QRB). The right axis indicates the percentage of improvement (p) over the traditional method, i.e., $(t_{tran} - t_{prop})/t_{tran}$.

5. Conclusions

In an exact region query, the candidate features retrieved by a spatial index in the filter step must be further examined in the refinement step. A new spatial index, named the QRB-tree index, is developed based on a QR-tree index to optimize the filter and refinement steps. In the filter step, a bucket is introduced for every grid in order to accelerate the

steps of loading and searching for the grids inside a query region. In the refinement step, the number of candidate features to be eliminated is reduced by limiting them to those whose MBR contains the corners of a query region. The performance of the QRB-tree index depends on the maximal grid level (L^0). To choose an acceptable value for L^0 , a bucket threshold pertaining to bucket loading and an auxiliary threshold pertaining to the index file location are proposed. Simulations show that the QRB-tree index runs faster than the QR-tree index for rough queries regardless of the size of the query region or density of the features. Real tests show that (a) the performance is similar when the query region is small, although the QRB-tree index outperforms the other two indices when the query region is large in the case of a rough query; (b) the QRB-tree index achieves the highest performance, followed by the QR-tree index and geohash index, when an exact query is performed; and (c) the new refinement method outperforms the traditional refinement method.

Although several improvements can be achieved using the QRB-tree index, a distributed version of this index is necessary to respond to the challenges posed by big data. Due to the nature of data partitioning in the QRB-tree index, the development of a distributed QRB-tree index is promising and will be considered in our future work. In addition, since the decomposition of grids in the QRB-tree index is not object-oriented, the bottleneck of the QRB-tree index may lie in the associated R-trees of the low-level grids, into which features are more likely to be registered. We will investigate this issue and attempt to solve it in future work.

Author Contributions: Conceptualization, Jieqing Yu, Yi Wei; methodology, Jieqing Yu, Yi Wei; validation, Yi Wei; formal analysis, Jieqing Yu, Yi Wei; data curation, Yi Wei, Qi Chu; writing—original draft preparation, Yi Wei; writing—review and editing, Jieqing Yu, Lixin Wu. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Key Research and Development Program of China (2018YFB0505304 and 2018YFC1503505) and the National Natural Science Foundation of China (41771416).

Data Availability Statement: The data and codes that support the findings of this study are available at <https://github.com/yujieqing/QRB-tree-index> (accessed on 13 October 2021).

Acknowledgments: We thank the Ordnance Survey of the United Kingdom for providing the dataset for testing. We thank Yariv Barkan and Oliver Keyes for providing the codes of the R-tree and geohash for the test. We also thank the anonymous reviewers, whose suggestions helped to enhance the quality of this paper.

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

References

1. Yao, X.; Li, G. Big spatial vector data management: A review. *Big Earth Data* **2018**, *2*, 108–129. [\[CrossRef\]](#)
2. Guo, H.; Liu, Z.; Jiang, H.; Wang, C.; Liu, J.; Liang, D. Big Earth Data: A new challenge and opportunity for Digital Earth's development. *Int. J. Digit. Earth* **2016**, *10*, 1–12. [\[CrossRef\]](#)
3. OGC. *OpenGIS® Implementation Standard for Geographic information—Simple Feature Access—Part 1: Common Architecture*; OGC 06-103r4; Open Geospatial Consortium: Rockville, MD, USA, 2011.
4. Mamoulis, N. Spatial Data Management. In *Synthesis. Lectures on. Data Management*; Morgan & Claypool: San Rafael, CA, USA, 2011; Volume 3, pp. 1–149. [\[CrossRef\]](#)
5. Kothuri, R.K.; Ravada, S. Efficient Processing of Large Spatial Queries Using Interior Approximations. In Proceedings of the 7th International Symposium, SSTD 2001, Redondo Beach, CA, USA, 12–15 July 2001; Advances in Spatial and Temporal Databases. pp. 404–421. [\[CrossRef\]](#)
6. Park, H.-H.; Cho, H.-J.; Chung, C.-W. Heuristic approach for early separated filter and refinement strategy in spatial query optimization. *J. Syst. Softw.* **2002**, *62*, 161–179. [\[CrossRef\]](#)
7. Kothuri, R.K.V.; Ravada, S.; Abugov, D. Quadtree and R-tree indexes in oracle spatial: A comparison using GIS data. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Madison, WI, USA, 4–6 June 2002; pp. 546–557. [\[CrossRef\]](#)
8. Nievergelt, J.; Hinterberger, H.; Sevcik, K.C. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.* **1984**, *9*, 38–71. [\[CrossRef\]](#)
9. Finkel, R.A.; Bentley, J.L. Quad trees a data structure for retrieval on composite keys. *Acta Inform.* **1974**, *4*, 1–9. [\[CrossRef\]](#)

10. Bartholdi, J.J.; Goldsman, P. Continuous indexing of hierarchical subdivisions of the globe. *Int. J. Geogr. Inf. Sci.* **2001**, *15*, 489–522. [[CrossRef](#)]
11. Ottoson, P.; Hauska, H. Ellipsoidal quadrees for indexing of global geographical data. *Int. J. Geogr. Inf. Sci.* **2002**, *16*, 213–226. [[CrossRef](#)]
12. Cho, W.; Choi, E. A basis of spatial big data analysis with map-matching system. *Clust. Comput.* **2017**, *20*, 2177–2192. [[CrossRef](#)]
13. Sakr, M. A data model and algorithms for a spatial data marketplace. *Int. J. Geogr. Inf. Sci.* **2018**, *32*, 2140–2168. [[CrossRef](#)]
14. Lei, Y.; Tong, X.; Zhang, Y.; Qiu, C.; Wu, X.; Lai, G.; Li, H.; Guo, C.; Zhang, Y. Global multi-scale grid integer coding and spatial indexing: A novel approach for big earth observation data. *ISPRS J. Photogramm. Remote. Sens.* **2020**, *163*, 202–213. [[CrossRef](#)]
15. Guttman, A. R-trees: A dynamic index structure for spatial searching. In Proceedings of the ACM management of data (SIGMOD), Boston, MA, USA, 18–21 June 1984; pp. 47–57.
16. Sellis, T.; Roussopoulos, N.; Faloutsos, C. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In Proceedings of the Thirteenth International Conference on Very Large Data Bases: 1987, 13th VLDB, Brighton, UK, 1–4 September 1987; pp. 507–518. [[CrossRef](#)]
17. Beckmann, N.; Kriegel, H.-P.; Schneider, R.; Seeger, B. The R*-tree: An efficient and robust access method for points and rectangles. *ACM SIGMOD Rec.* **1990**, *19*, 322–331. [[CrossRef](#)]
18. Kamel, I.; Faloutsos, C. Hilbert R-tree: An improved R-tree using fractals. In Proceedings of the 20th International Conference on Very Large Databases, Santiago, Chile, 12–15 September 1993; pp. 500–509.
19. Arge, L.; De Berg, M.M.; Haverkort, H.H.; Yi, K. The priority R-tree: A Practically Efficient and Worst-Case Optimal. *R-tree. ACM Trans. Algorithms* **2008**, *4*, 1–30. [[CrossRef](#)]
20. Silva, Y.N.; Xiong, X.; Aref, W.G. The RUM-tree: Supporting frequent updates in R-trees using memos. *VLDB J.* **2009**, *18*, 719–738. [[CrossRef](#)]
21. Yang, Y.; Bai, P.; Ge, N.; Gao, Z.; Qiu, X. LAZY R-tree: The R-tree with lazy splitting algorithm. *J. Inf. Sci.* **2019**, *46*, 243–257. [[CrossRef](#)]
22. Singh, H.; Bawa, S. A Survey of Traditional and MapReduce-Based Spatial Query Processing Approaches. *SIGMOD Rec.* **2017**, *46*, 18–29. [[CrossRef](#)]
23. Goyal, P.; Challa, J.S.; Kumar, D.; Bhat, A.; Balasubramaniam, S.; Goyal, N. Grid-R-tree: A data structure for efficient neighborhood and nearest neighbor queries in data mining. *Int. J. Data Sci. Anal.* **2020**, *10*, 25–47. [[CrossRef](#)]
24. Lee, K.Y.; Kang, J.J.; Kim, J.J.; Choi, G.S.; Lee, Y.D.; Cho, S.Y.; Oh, S.J. Indexing method for moving sensor node retrieval. *Int. J. Sens. Netw.* **2014**, *15*, 238–245. [[CrossRef](#)]
25. Fu, Y.-C.; Hu, Z.-Y.; Guo, W.; Zhou, D.-R. QR-tree: A hybrid spatial index structure. In Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693), Xi'an, China, 5 November 2003. [[CrossRef](#)]
26. Huang, B.; Wu, Q. A Spatial Indexing Approach for High Performance Location Based Services. *J. Navig.* **2007**, *60*, 83–93. [[CrossRef](#)]
27. Wang, Y.; Zhu, Y.; Sun, H. Study of Spatial Data Index Structure Based on Hybrid Tree. In *Knowledge Engineering and Management*; Springer: Berlin/Heidelberg, Germany, 2011; Volume 123, pp. 559–565. [[CrossRef](#)]
28. Yang, Z.X.; Hao, Z.X. Spatial Join Queries Based on QR-tree. *Adv. Mater. Res.* **2011**, *187*, 752–757. [[CrossRef](#)]
29. Mao, H.; Bian, F. Design and Implementation of QR+Tree Index Algorithms. In Proceedings of the 2007 International Conference on Wireless Communications, Networking and Mobile Computing, Shanghai, China, 21–25 September 2007; pp. 5987–5990. [[CrossRef](#)]
30. Phan, T.K.; Jung, H.; Youn, H.Y.; Kim, U.M. QR*-Tree: An Adaptive Space-Partitioning Index for Monitoring Moving Objects. *J. Inf. Sci. Eng.* **2017**, *33*, 385–411. [[CrossRef](#)]
31. Guo, J.; Guo, W.; Zhou, D. Indexing of Constrained Moving Objects for Current and Near Future Positions in GIS. In Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06), Hangzhou, China, 20–24 June 2006. [[CrossRef](#)]
32. Hjaltason, G.R.; Samet, H. Speeding up construction of PMR quadtree-based spatial indexes. *VLDB J.* **2002**, *11*, 109–137. [[CrossRef](#)]
33. Hohl, A.; Casas, I.; Delmelle, E.; Tang, W. Hybrid Indexing for Parallel Analysis of Spatiotemporal Point Patterns. In Proceedings of the 9th International Conference on Geographic Information Science, Montreal, QC, Canada, 27–30 September 2016. [[CrossRef](#)]
34. Yang, J.; Huang, X. A Hybrid Spatial Index for Massive Point Cloud Data Management and Visualization. *Trans. GIS* **2014**, *18*, 97–108. [[CrossRef](#)]
35. Gu, W.; Wang, J.; Shi, H.; Liu, Y. Research on a hybrid spatial index structure. *J. Colloid Interface Sci.* **2011**, *11*, 3972–3978.
36. 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](#)]
37. Du Mouza, C.; Litwin, W.; Rigaux, P. SD-Rtree: A Scalable Distributed Rtree. In Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering, Istanbul, Turkey, 15–20 April 2007; pp. 296–305. [[CrossRef](#)]
38. Malensek, M.; Pallickara, S.; Pallickara, S. Evaluating Geospatial Geometry and Proximity Queries Using Distributed Hash Tables. *Comput. Sci. Eng.* **2014**, *16*, 53–61. [[CrossRef](#)]
39. Elashry, A.; Shehab, A.; Riad, A.M.; Aboul-Fotouh, A. 2DPR-tree: Two-Dimensional Priority R-tree Algorithm for Spatial Partitioning in SpatialHadoop. *ISPRS Int. J. Geo-Inf.* **2018**, *7*, 179. [[CrossRef](#)]

-
40. Xia, J.; Huang, S.; Zhang, S.; Li, X.; Lyu, J.; Xiu, W.; Tu, W. DAPR-tree: A distributed spatial data indexing scheme with data access patterns to support Digital Earth initiatives. *Int. J. Digit. Earth* **2020**, *13*, 1656–1671. [[CrossRef](#)]
 41. Han, D.; Stroulia, E. HGrid: A Data Model for Large Geospatial Data Sets in HBase. In Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, 28 June–3 July 2013; pp. 910–917. [[CrossRef](#)]
 42. Park, S.-Y.; Bae, H.-Y. A Distributed Spatial Index for Time-Efficient Aggregation Query Processing in Sensor Networks. In Proceedings of the Computational Science—ICCS, Singapore, 9–12 May 2005; pp. 405–410. [[CrossRef](#)]
 43. Bianchi, S.; Felber, P.; Potop-Butucaru, M.G. Stabilizing Distributed R-trees for Peer-to-Peer Content Routing. *IEEE Trans. Parallel Distrib. Syst.* **2010**, *21*, 1175–1187. [[CrossRef](#)]
 44. Feng, J.; Ren, F.; Tang, Z. Hadoop-based QR-tree index. *Comput. Eng. Des.* **2013**, *12*, 4231–4236. [[CrossRef](#)]
 45. Singh, H.; Bawa, S. An improved integrated Grid and MapReduce-Hadoop architecture for spatial data: Hilbert TGS R-tree-based IGSIM. *Concurr. Comput. Pract. Exp.* **2019**, *31*, e5202. [[CrossRef](#)]
 46. Morton, G.M. A computer oriented geodetic data base and a new technique in file sequencing. *IBM Ger. Sci. Symp. Ser.* **1966**, 294–897.
 47. Huffman, C. *Storage*; Syngress: Rockland, MA, USA, 2015.
 48. Varalakshmi, M.; Kesarkar, A.P.; Lopez, D. High-performance implementation of a two-bit geohash coding technique for nearest neighbor search. *Concurr. Comput. Pract. Exp.* **2020**, *33*, e6029. [[CrossRef](#)]