A new enhancement to the R-tree node splitting

Amer F. Al-Badarneh, Qussai Yaseen and Ismail Hmeidi

School of Computer and Information Technology, Jordan University of Science & Technology, Irbid, Jordan

Abstract.

The performance of spatial queries depends mainly on the underlying index structure used to handle them. R-tree, a well-known spatial index structure, suffers largely from high overlap and high coverage resulting mainly from splitting the overflowed nodes. Assigning the remaining entries to the underflow node in order to meet the R-tree minimum fill constraint (Remaining Entries problem) may induce high overlap or high coverage. This is done without considering the geometric features of the remaining entries and this may cause a very non-optimized expansion of that particular node. The proposed solution to this problem distributes rectangles as follow: (1) Assign \( m \) entries to the first node, which are nearest to the first seed. (2) Assign other \( m \) entries to the second node, which are nearest to the second seed. (3) Assign the remaining entries one by one to the nearest seed. Several experiments on real data, as well as synthetic data, show that the proposed splitting algorithm outperforms the efficient version of the original R-tree in terms of query performance.

Keywords: node splitting; R-tree; spatial database; spatial indexing

1. Introduction

Spatial data management has been an active area of intensive research for more than two decades. Spatial data have grown everywhere; from CAD [1] and geographical databases [2] to multimedia systems [3] and image processing [4]. Spatial databases store multidimensional objects such as points, lines, rectangles and polygons. To support spatial objects efficiently in traditional database management systems, several issues should be taken into consideration including spatial data models, indexing techniques, and efficient query processing. To handle spatial queries efficiently, a competent spatial index structure is needed. The major problem of designing spatial indices, however, is that no total ordering among spatial objects exists that preserves their spatial proximity [5] and there is no one-dimensional index structure (e.g. B-tree [6] and its variants [7]) that can handle spatial objects efficiently [8].

One of the most significant index structures proposed for indexing spatial data is the R-tree [9]. The main idea of the R-tree is to split the space with hierarchically nested and possibly overlapping subspaces. The hierarchical structure of the R-tree index is height balanced (i.e. all leaf nodes appear at the same level) similar to that of B-tree index. The structure is dynamic (i.e. updates,
inserts and deletes of objects can be intermixed with searches without the need to reorganize the tree structure) and guarantees that the space utilization is at least 50%.

A primary reason for the popularity of R-trees is their supporting for different types of spatial queries, such as topological queries [10, 11] (e.g. covering, overlapping and crossing), directional queries [12] (e.g. north, east, and north east), and metric (distance) queries (e.g. k-nearest) [13]. Also, the simplicity of the structure and its resemblance to the B-tree allowed developers to easily incorporate the structure into existing database management systems to support spatial query processing [14].

Each spatial object in the R-tree is approximated by a minimum bounding rectangle (MBR). For a two-dimensional space, each MBR is represented by a 4-tuple entry \((x_l, x_h, y_l, y_h)\), where \(x_l\) and \(x_h\) represent the low and high values of the \(X\) dimension (i.e. the extend of MBR along the \(X\) dimension), while \(y_l\) and \(y_h\) represent the extend of MBR along the \(Y\) dimension. Figure 1 illustrates an example of a two-dimensional spatial object and its MBR.

Each leaf node of the R-tree contains spatial object entries; each entry contains the object address and its MBR coordinates. Each entry within a non-leaf node contains a child-node identifier and coordinates of the smallest MBR that spatially encompasses all the MBRs of the child-node’s entries. Each node in the R-tree, except the root, has a capacity (fan-out) between \(m\) (the minimum number of entries in a node) and \(M\) (the maximum number of entries that will fit in one node). \(M\) depends on the machine’s disk page size and the number of space’s dimensions. The value of \(m\) should be at most \(\lceil M/2 \rceil\); this is called minimum fan-out constraint. To guarantee good space utilization and query performance, the value of \(m\) is usually = 40% relatively to \(M\) [15]. Figure 2 shows a set of rectangles and their corresponding R-tree structure where \(M = 3\).

Searching in the R-tree is similar to that in B-tree. However, the performance of the search operation is affected mainly by two parameters: coverage and overlap. Coverage of a level of an R-tree is defined as the total area of all rectangles associated with the nodes at that level. Overlap of a level of an R-tree is defined as the total area contained within two or more nodes of that level. High overlap or high coverage would produce poor R-tree structure, because multiple paths may be traversed in order to answer a spatial query and this will decrease the search performance. Obviously, minimization of both overlap and coverage is crucial to R-tree query performance [16]. In some cases, optimizing one of them may affect the other in a negative way. Figure 3 illustrates this point, where a zero overlap is achieved with large coverage (Figure 3(a)), while minimal coverage is achieved with large overlap (Figure 3(b)).

Ever since its introduction in 1984, many variants of the original R-tree have been developed to improve its performance, mainly by minimizing overlap, coverage or both. These variants differ primarily in the techniques used to split an overflowing node during insertion, striving for better
overall behaviour (e.g. reduction in coverage and overlap). To have overlap area or coverage area optimized, an efficient splitting method is essentially needed. Splitting occurs when adding a new entry to a full node containing \( M \) entries. To split the overflowed node, the \( M + 1 \) entries should be distributed between two nodes, preserving the minimum fan-out constraint. The distribution is done in a way that makes it as unlikely as possible that both new nodes will need to be examined on subsequent searches. Since, the decision whether to visit a node depends on whether its covering rectangle overlaps the search area, the total area of the two covering rectangles should be minimized.

The optimal distribution is the one that produces two nodes that have minimum overlap and/or coverage with optimal splitting cost. Guttman in his original paper [9] proposed three different methods to handle splitting, and distinguished them by their time complexity: linear, quadratic, and exponential. Among them, the quadratic method is the one that achieves the best compromise between splitting cost and bipartition optimality. All three proposed methods are based on the concept of least enlargement to minimize the coverage area of the bounding rectangles of the resulting two new nodes.

Other splitting methods have been proposed by using different optimization criteria. The R*-tree [17] is proposed to achieve zero overlap by inserting the new entry into all nodes that it intersects. However, in the R*-tree the data object can be found in more than one leaf node. The R'-tree [12]...
The quadratic split algorithm of the R-tree suffers from some problems. One of them is called the ‘Remaining Entries’ problem: if one of the new nodes has reached the maximum number of entries \(M - m + 1\), all remaining entries are assigned to the other node in order to reach the minimum fill \(m\). This is done without considering the geometric features of the remaining entries and this may cause a very non-optimized splitting of that particular node. Figure 4 illustrates an example showing the ‘Remaining Entries’ problem. The result is a split with uneven distribution of the entries reducing the storage utilization and increasing the coverage area (Figure 4(b)) [12].

In this paper, a solution to the ‘Remaining Entries’ problem of quadratic R-tree is proposed. The proposed solution distributes rectangles as follows:

1. Assign \(m\) entries to the first node, which are nearest to the first seed.
2. Assign other \(m\) entries to the second node, which are nearest to the second seed.
3. Assign the remaining entries one by one to the nearest seed.

As shown in Figure 4(c), the two nodes that are created using our solution have less coverage and more space utilization.

The paper is organized as follows. Section 2 introduces a summary of some variants of R-tree splitting techniques as a background of the work. Section 3 explains the proposed splitting technique. Section 4 provides the experiments performed using the proposed technique and the quadratic version of R-tree. Section 5 concludes and presents the future work.

2. Related work

The most crucial part of the R-tree construction is the node splitting procedure of insertion algorithm. Inserting a new entry to the R-tree is similar to insertion in a B-tree. The appropriate leaf node, say \(N\), is selected by traversing the tree top-down starting from the root and choosing at each
level the child node whose MBR has to be enlarged least to occupy the new entry. If there is room in \( N \) (i.e., \( N \cdot \text{size} < M \)), then the new entry is added to \( N \), otherwise a split takes place.

Node splitting is invoked when inserting a new entry to a full node containing \( M \) entries. The collection of \( M + 1 \) entries should be distributed between two nodes. The total area of the two nodes after a split should be minimized. In [7], three different methods are proposed to handle node splitting: linear, quadratic and exponential. Among them, the quadratic method is the one that achieves the best compromise between splitting cost and bipartition optimality. In the following, the quadratic algorithm is analyzed and discussed.

2.1. R-tree quadratic split

The quadratic splitting algorithm of R-tree, shown in Figure 5, aims to find a small area split, but it does not guarantee to find the split with the smallest area possible. The algorithm starts first by invoking \( \text{QPickSeeds} \) algorithm, shown in Figure 6, to select two entries (seeds) to initiate the two new nodes, say \( N_1 \) and \( N_2 \). This is done by examining all the entries of the overflowing node and finding the pair of entries that would waste the most area were they to be inserted in the same node. This is determined by subtracting the sum of the areas of the two entries from the area of the covering bounding box. These two entries are picked to be the first elements of the two new nodes \( N_1 \) and \( N_2 \). The \( \text{QPickSeeds} \) algorithm takes \( O(M) \), since the for loop and its body, shown in Line 2 and Line 3 respectively, of \( \text{QPickSeeds} \) algorithm are repeated \( O(M) \) times each.

The remaining entries of the overflowing node are selected one by one to be put to one of the two nodes using \( \text{QDistribute} \) algorithm, shown in Figure 7. The entry is selected using \( \text{PickNext} \) algorithm, shown in Figure 8. The entry \( e \), whose addition maximizes the difference in coverage between the bounding boxes associated with \( N_1 \) and \( N_2 \), is added to the node whose coverage is minimized by the addition. This process is reapplied to the remaining entries until all entries of the overflowing node are assigned to either nodes or one of the nodes \( N_1 \) or \( N_2 \) has reached the maximum number of entries \( (M - m + 1) \). If the later constraint takes place, all remaining entries are assigned to the other node in order to reach the minimum fill \( m \). The cost of \( \text{PickNext} \) is \( O(M) \) and since \( \text{PickNext} \) is invoked at most \( M \) times within \( \text{QDistribute} \) algorithm then the time complexity of the \( \text{QDistribute} \) algorithm is \( O(M^2) \). Therefore, the time complexity of the quadratic splitting algorithm of R-tree is \( O(M^2) \).

2.2. Other R-tree splitting techniques

R+-tree [17] was proposed to achieve zero overlap among the internal nodes. The space is divided into disjoint sub-regions and each entry is associated with each leaf node that it intersects. That means there may be several paths starting at the root to the same entry. The trade-off is that more
space is required because of the duplicate entries and thus the height of the R*-tree may be greater than the original R-tree. However, the logarithmic increase in the height of the R*-tree is a consequence which may still be better than searching multiple shorter paths in the original R-tree when an entry overlaps more than one internal node at the same level. Another related drawback is that deleting an entry may lead to deleting more than one entry from the leaf nodes.

R*-tree [12] is another version of the R-tree. Unlike R*-tree, it permits overlap among its nodes, but it applies a more complex insertion and split algorithms to reduce both coverage of internal nodes
and overlap for leaf nodes. The insertion algorithm is based on the forced reinsert concept (i.e. some of the overflowing node entries are deleted and reinserted). This has the effect of producing more well-clustered groups of entries in nodes, reducing node coverage. Furthermore, actual node splits are often postponed, causing average node occupancy (storage utilization) to rise. The drawback of R*-tree is that its implementation cost is higher than for the other R-trees. Even that reinsertion may increase the storage utilization; but the overhead of reinsertion can be fairly expensive [24].

Hilbert R-tree [18] was proposed to achieve high space utilization by minimizing the area and perimeter of the resulting MBRs. They employ a flexible deferred splitting mechanism to increase the space utilization. Every node has a well-defined set of sibling nodes. By adjusting the split policy the Hilbert R-tree can achieve a degree of space utilization as high as is desired. This is done by proposing an ordering on the R-tree nodes. Hilbert R-tree sorts data objects according to the Hilbert value of the centre of their rectangles (i.e. MBR). Given the ordering, every node has a well-defined set of sibling nodes; thus, deferred splitting can be used. By adjusting the split policy, the Hilbert R-tree can achieve as high a utilization as desired. To the contrary, other R-tree variants have no control over the space utilization.

Huang et al. proposed Compact R-tree [20] that significantly improves storage utilization, with no penalty on query performance. The main idea of the insertion algorithm of Compact R-tree is to select M entries, out of the M + 1 entries of an overflowed node N, to remain in this node, under the constraint that the resulting N’s MBR is the minimum possible. Then the remaining entry is inserted to a sibling node that has available space and whose MBR is enlarged as little as possible. Experimental results presented in [20] have shown that the storage utilization of the Compact R-tree is above 95%. The resulting R-tree will have fewer nodes, which leads to less construction time.

Brakatsoulas et al. [16] proposed the cR-tree, which treats the node splitting process as a typical clustering problem. The main idea of cR-tree is to distribute the M + 1 entries of the overflowed node into more than two nodes (clusters) in order to maximize the similarity of spatial objects within each cluster (node) and minimize the similarity of spatial objects across clusters, hence minimizing overlap and coverage. The experimental results reported in [16] have shown that the cR-tree query performance to be competitive with the R*-tree and much better than that of the R-tree.

Ang and Tan [25] proposed the Linear R-tree that has a linear time node splitting algorithm. The splitting algorithm distributes the entries of the overflowed node into two new nodes using three different criteria. First, the algorithm distributes the entries between the two nodes as evenly as possible, whereas the second criterion is the minimization of the overlap between them. Finally, the third criterion is the minimization of the total coverage. As a result, all entries within an overflowed node are pushed to either side in each dimension such that the entries of the two new nodes end up in the furthest separation as possible with less overlap.
3. The proposed splitting algorithm

The quadratic splitting approach of the original R-tree consists of two main algorithms: *QP*ickSeeds to select the two seeds and *QD*istribute to put the remaining entries into the two nodes. In the following, the proposed enhancements of these two algorithms are presented.

3.1. The Linear *P*ickSeeds algorithm

The *QP*ickSeeds algorithm of the original quadratic R-tree requires two iterations through the node entries in order to select the two seeds. The cost is quadratic in $M$. Here we propose a linear-cost algorithm for selecting the two seeds. In our algorithm *LP*ickSeeds (shown in Figure 9), the entry whose MBR has the *lowest* sum of the low $x$ and $y$ values is selected as the first seed. The second seed is selected by picking the entry whose MBR has the highest sum of the highest $x$ and $y$.

$$
1. \text{lowest} = N.e[i].x_l + N.e[i].y_l \\
2. \text{highest} = N.e[i].x_h + N.e[i].y_h \\
3. s = t = N.e[i] \\
4. \text{for } i = 2 \text{ to } M+1 \text{ do} \\
5. \text{lowxy} = N.e[i].x_l + N.e[i].y_l \\
6. \text{if } \text{lowxy} < \text{lowest} \text{ then } \text{lowest} = \text{lowxy}, \quad s = N.e[i] \\
7. \text{highxy} = N.e[i].x_h + N.e[i].y_h \\
8. \text{if } \text{highxy} > \text{highest} \text{ then } \text{highest} = \text{highxy}, \quad t = N.e[i] \\
9. \text{return } s \text{ and } t
$$

Fig. 9. LPickSeeds algorithm.

![Fig.9. LPickSeeds algorithm.](image)

Fig. 10. Picking seeds using the quadratic (B & C entries) and the proposed (A & C entries) approaches.

3. The proposed splitting algorithm

The quadratic splitting approach of the original R-tree consists of two main algorithms: *QP*ickSeeds to select the two seeds and *QD*istribute to put the remaining entries into the two nodes. In the following, the proposed enhancements of these two algorithms are presented.

3.1. The Linear *P*ickSeeds algorithm

The *QP*ickSeeds algorithm of the original quadratic R-tree requires two iterations through the node entries in order to select the two seeds. The cost is quadratic in $M$. Here we propose a linear-cost algorithm for selecting the two seeds. In our algorithm *LP*ickSeeds (shown in Figure 9), the entry whose MBR has the *lowest* sum of the low $x$ and $y$ values is selected as the first seed. The second seed is selected by picking the entry whose MBR has the *highest* sum of the high $x$ and $y$ values. For example, *QP*ickSeeds chooses entries $B$ and $C$ as the two seeds of the spatial objects shown in Figure 10, while *LP*ickSeeds chooses $A$ and $C$ as the two seeds. In some cases, the two algorithms may select two different pairs of seeds. The main contribution of our *LP*ickSeeds algorithm is to select two seeds as good as the quadratic approach does but with linear time complexity. Figure 11 shows the resulting two nodes with high overlap and high coverage. On
the other hand, Figure 12 shows the splitting using the proposed approach with zero overlap and less coverage. Sometimes, we may have a node that has both the lowest sum of \( x_l \) and \( y_l \) values and the highest sum of \( x_h \) and \( y_h \) values. To break this tie, the node that has the second lowest sum of \( x_l \) and \( y_l \) values is chosen.

3.2. The New Distribute algorithm

One of the problems that the quadratic splitting algorithm suffers from is the ‘Remaining Entries’ problem. This occurs when one of the nodes has reached the maximum number of entries (\( M - m + 1 \)). All remaining entries are assigned to the other node in order to reach the minimum fill (\( m \)). This is done without considering the geometric features of the remaining entries and this may produce a large amount of coverage and cause a very non-optimized expansion of that particular node. The proposed enhancement of this algorithm overcomes the above problem. The new algorithm NDistribute, shown in Figure 13, uses the following steps:

**Step 1:** Using the first seed \( s_i \) (the one that has the lowest sum of \( x_l \) and \( y_l \)):

- For each entry \( e \) in the overflowed node \( N \), compute the distance \( d \) between the lower left corner of \( s_i \) \((s_i.x_l, s_i.y_l)\) and the upper right corner of \( e \) \((e.x_r, e.y_r)\).
- Sort entries ascending according to the distance \( d \).
- Assign first \( m \) entries in \( N \) to node \( N_1 \).

**Step 2:** Using the second seed \( s_j \) (the one that has the highest sum of \( x_h \) and \( y_h \)):

- For each remaining entry \( e \) in \( N \), compute the distance \( d \) between the upper right corner of \( s_j \) \((s_j.x_r, s_j.y_r)\) and the lower left corner of \( e \) \((e.x_l, e.y_l)\).
Sort entries ascending according to the distance $d$.

Assign first $m$ entries in $N$ to node $N_1$.

**Step 3:** Assign the remaining $M - 2m + 1$ entries one by one to the nearest node ($N_1$ or $N_2$) using the distance measure used in Steps 1 and 2.

The proposed algorithm produces more improved R-tree structure than original quadratic approach, since it resolves the problem of 'Remaining Entries' that could produce large overlap or coverage (see Figures 10–12). Using the given distance function, the proposed algorithm assigns to the first seed its nearest $m$ entries, and to the second seed its nearest $m$ entries, which should produce two new nodes with low coverage. By doing this, we eliminate the possibility of assigning node entries without considering the high coverage area that may be produced. The rest of the entries are assigned to their nearest seeds. The most dominant statement in the NDistribute algorithm is the sorting. Instead of using $O(M \log M)$ sorting algorithm, a min heap is used to store the node entries and then do $m \text{DeleteMin}$, with a total cost of $O(m \log M)$. As a result, the time complexity of the proposed splitting algorithm is $O(n \log M)$, while the time complexity of the original splitting algorithm is $O(M^2)$.

### 4. Experimental results

In this section, we describe the results of our experimental study of the performance of the proposed R-tree (we will refer to it as NR-tree) and give empirical evidence for practicality of NR-tree when compared to the quadratic version of the original R-tree (QR-tree). We used the quadratic version of
R-tree in our comparisons since NR-tree is an enhancement of QR-tree and QR-tree has been used by many researchers as a model to measure the performance of their spatial index structures. Below we discuss the experimental setup, datasets used, and the time results of query experiments.

### 4.1. Experimental setup

All experiments were run on a Core 2 Duo/2.5 GHz machine with 2 GB memory running Windows XP. To implement the algorithms of the two R-trees, we used SaIL [26], which is a Java-based open source framework for implementing disk-page R-tree and its variants. Since the CPU time required to process pages is negligible, performance is measured as the average number of disk page accesses in answering a workload of 10 range spatial queries. The number of disk pages accessed to answer a query is incremented by one each time a disk page is read from persistent storage. If a page is not located in the memory cache, it is fetched (read) from hard disk to the cache. Reading a node from cache is not counted as a page access. The disk page sizes investigated in this study were 0.5 KB and 1 KB. We select a relatively small page size to simulate practical situations, where the database is expected to be considerably larger.

Values in all the dimensions are represented by 4-byte integers. In our experiments, each node is represented by: (i) node level, (ii) used space (how much space is used up into this node), and (iii) an array that stores the node entries. The size of this array is node capacity (M). Each node entry occupies $8d + 4$ bytes, which includes $8d$ bytes for the MBR coordinates and 4 bytes for the page number where the entry is stored. $d$ is the space dimension. And hence, the disk page (node) size equals to $M \times (8d + 4) + 8$. In the 2D space, the node capacity $M$ for 0.5 KB and 1 KB pages is 25 and 50 entries, respectively.

### 4.2. Datasets used

We deployed both real-life and synthetic two-dimensional data in our experiments. As the real-life data, we used the TIGER/Lines data distributed by the US Bureau of Census [27]. These data are the standard benchmark data used in spatial databases and consist of line segments that represent roads, rail lines, rivers, etc. for US states. We chose to experiment with the road line segments from two datasets obtained from [28]. For each dataset, we used the bounding boxes of the line segments as our input rectangles. The first dataset had 98,450 rectangles of California and the second dataset had 128,000 rectangles of four US states: Iowa, Kansas, Missouri and Nebraska. We use CA and IKMN to refer to these two datasets, respectively.

To investigate how the proposed R-tree performs on more extreme datasets than the Tiger data, we generated two synthetic datasets. Each of these synthetic datasets consists of 100,000 rectangles in the unit square $[0, 1]^2$. MATLAB was used to generate the centres of the synthetic data rectangles. We designed the first synthetic dataset to investigate how well the R-trees handle rectangles of uniform distribution. The rectangle centres were uniformly distributed. The maximum rectangle area was chosen to be 0.001 of the space. The average rectangles area was 0.00026 of the space. When generating the datasets, we discarded rectangles that were not completely inside the unit square. The dataset size is 100,000 rectangles. The second synthetic dataset was designed to investigate how the R-trees handle rectangles with normal distribution. The rectangle centres were normally distributed. The maximum rectangle area was chosen to be 0.001 of the space. The average rectangles area was 0.00024 of the space. The dataset size is 100,000 rectangles. We use UNIFORM and NORMAL to refer to these two datasets, respectively.

### 4.3. Experimental results

The proposed R-tree (NR-tree) was implemented and several experiments were conducted to demonstrate its performance superiority over the quadratic version of R-tree (QR-tree). After building the two R-tree variants, we present experiments to assess their query performance. We used
different query window sizes, range from 0.5% to 90% of the total space area. For each query window size, we randomly created 10 square queries and computed their page accesses average. We used line charts to present the results. The Y-axis represents the performance ratio value, which is (page accesses retrieved using QR-tree)/(page accesses retrieved using NR-tree). The critical value is one, and if this ratio is greater than one, it means that it takes more time searching the QR-tree to find results than searching the NR-tree. The X-axis represents the different query sizes used.

We first performed experiments using real-life data. Figure 14 shows the experimental results using CA dataset. We noticed that NR-tree outperforms QR-tree for all query sizes greater than 1% and 4% using page sizes 0.5 K and 1 K, respectively. The NR-tree achieves up to 9% saving in response time over QR-tree. The experimental results using IKN dataset are presented in Figure 15. Here also, NR-tree outperforms QR-tree for query sizes greater than 1% and achieving up to 15% time saving using 0.5 K page size and up to 11% saving using 1 K page size.
We did the same experiments over the synthetic data. Figure 16 and Figure 17 show the performance gain ratio of NR-tree over QR-tree using UNIFORM and NORMAL datasets respectively. Here NR-tree outperforms QR-tree for query sizes over 50% and 70% using UNIFORM dataset with 0.5 K and 1 K page sizes respectively. Also, NR-tree outperforms QR-tree for query sizes over 8% and 10% using NORMAL dataset with 0.5 K and 1 K page sizes respectively. We noticed that NR-tree achieves 16% saving in response time over QR-tree using UNIFORM dataset and 32% time saving using NORMAL dataset with 0.5 K page size.

5. Conclusion

In spatial index structures like the R-tree, the overlap and coverage should be minimized. The R-tree’s quadratic splitting algorithm tries to achieve this goal but it suffers from some problems. One of them is called the ‘Remaining Entries’ problem: if one of the new nodes has reached the
maximum number of entries \((M - m + 1)\), all remaining entries are assigned to the other node in order to reach the minimum fill \((m)\). This is done without considering the geometric features of the remaining entries and this may cause a very non-optimized splitting of that particular node. In this paper, we have presented an efficient approach for handling R-tree node splits. The proposed algorithm is an enhancement of the quadratic version of the original R-tree. The main goal of the proposed approach is to handle the ‘Remaining Entries’ problem that may occur using the quadratic R-tree. The proposed solution distributes rectangles as follows:

1. Assign \(m\) entries to the first node, which are nearest to the first seed.
2. Assign other \(m\) entries to the second node, which are nearest to the second seed.
3. Assign the remaining entries one by one to the nearest seed.

Fig. 16. Query performance ratio varying the query size on the UNIFORM dataset.
We have performed several experiments using real, as well as synthetic data, different query sizes, and different disk page sizes. The results show that the proposed splitting algorithm has good improvements over the efficient version of the original R-tree in terms of query performance. The degree of improvement depends on the query size and disk page size: the higher query size and the smaller page size, the better performance of our approach. Furthermore, we noticed that, for small size queries, the proposed approach does not do well, because the main goal of the proposed approach is to reduce the coverage without taking the overlap produced into account. When the query size is small, the query will not cover a large area, and hence the effect of coverage will be minimal. How to process partial match queries and best match queries are future research topics, where partial match query means to report all data objects which are located in a specific line, and best match query means to find the nearest neighbour of a specific data object.
References


