A new double sorting-based node splitting algorithm for R-tree

Alexander Korotkov
National Research Nuclear University MEPhI
31 Kashirskoe shosse
Moscow, Russian Federation
Email: aekorotkov@gmail.com

***Abstract*—A storing of spatial data and processing of spatial queries are important tasks for modern databases. The execution efficiency of spatial query depends on underlying index structure. R-tree is a well-known spatial index structure. Currently there exist various versions of R-tree, and one of the most common variations between them is node splitting algorithm. The problem of node splitting in one-dimensional R-tree may seem to be too trivial to be considered separately. One-dimensional intervals can be split on the base of their sorting. Some of the node splitting algorithms for R-tree with two or more dimensions comprise one-dimensional split as their part. However, under detailed consideration, existing algorithms for one-dimensional split do not perform ideally in some complicated cases. This paper introduces a novel one-dimensional node splitting algorithm based on two sortings that can handle such complicated cases better. Also this paper introduces node splitting algorithm for R-tree with two or more dimensions that is based on the one-dimensional algorithm mentioned above. The tests show significantly better behavior of the proposed algorithms in the case of highly overlapping data.**

I. Introduction

Spatial data processing is an important task for modern databases. Since the volume of information in databases increases continuously, the database management systems (DBMS) need spatial index structures in order to handle spatial queries efficiently. The problem of spatial indexes is that there is no ordering which reflects proximity of spatial objects [5]. This is why B-tree [3] can not handle spatial object efficiently. R-tree [7] is the most well-known index structure for spatial data. R-tree is a height balanced tree like B-tree, which hierarchically splits space into possibly overlapping subspaces. Spatial objects in R-tree are approximated by minimal bounding rectangles (MBRs), see figure 1. Leaf node entry of R-tree contains MBR of spatial object and a reference to the corresponding database object. An entry of non-leaf node of R-tree contains reference to the child node and MBR of all rectangles in child node. Since the rectangles of a same node of R-tree can overlap, exact match query may lead to multipath tree scan. This forms significant difference of R-tree from such data structures as B-tree. The number of query paths and, in turn, the number of node accesses of non-exact match query also strongly depends on degree of rectangle overlap. R-tree was originally designed for access to multidimensional data, but it is also applied for one-dimensional intervals [10].

The quality of R-tree strongly depends on the node splitting algorithm. The task of node splitting is to split entries of the overflowed node into two groups which will form two new nodes. Node splitting algorithm substantially determines the area and degree of overlap of the tree rectangles. In turn these parameters determine the probability of multipath queries. The following parameters can be used in order to estimate the quality of a node splitting:

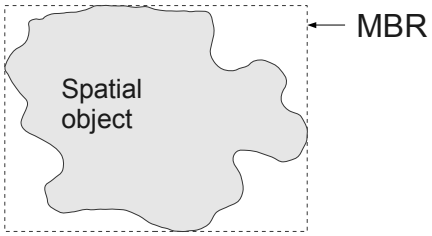

Fig. 1. MBR illustration

- The overlap of bounding rectangles. The smaller overlap of entry rectangles leads to the smaller probability of multipath queries.
- The coverage of bounding rectangles. The coverage of a split is a total area of bounding rectangles. In general smaller coverage leads to the smaller probability of multipath queries when query area is relatively large [1].
- Storage utilization. As the measure of storage utilization, a ratio between a numbers of entries in the smaller group and the greater group can be used. Typically, constraint is imposed on this parameter, i.e., the minimal number of entries in the resulting node m is defined. Restriction of this parameter is very reasonable, but this parameter can also be an optimization target. The higher ratio leads to the smaller tree balancing during construction. In turn, this influences the tree quality.

The illustration of dillemma between less overlap and less coverage is given on figure 2.

The paper is organized as follows. Section II describes node splitting algorithms which currently exist. Section III introduces double sorting-based one-dimensional node splitting algorithm and its generalization for multidimensional case. Section IV provides the experimental comparison of the proposed algorithm with other existing algorithms. Section V is a conclusion.

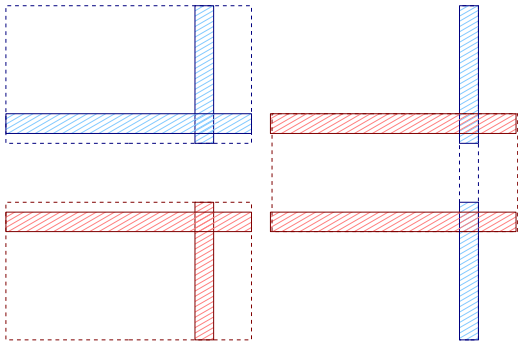

Fig. 2. Illustration of overlap vs. coverage dilemma

II. Related work

Originally Guttman in [7] introduced three node splitting algorithms:

- Exponential algorithm. This algorithm searches for global minimum of the area covered by rectangles by the enumerations of all possible splits. This method is too CPU expensive, because it requires exponential time.
- Quadratic algorithm. This algorithm consists of two steps. At the first step, two seeds of two resulting groups are selected. The seeds are selected as the rectangles that have maximal difference between their MBR area and their own area. At the second step, all other rectangles sequentially join some of the groups. Each time the rectangle for which the increase of MBR area due to its joining to one of the groups is maximal joins the group which MBR area increases less.
- Linear algorithm. This algorithm is similar to quadratic one, but it has two differences that make it linear. At first, seeds are selected along the axis that allows avoiding comparison of each pair of rectangles. The second is that rectangles join the groups in arbitrary order.

In [6] Green's algorithm was proposed. This algorithm is similar to Guttman's linear algorithm, but it uses sorting along the chosen axis and splitting entries at halves between the groups according to the sorting.

In [4] R*-tree splitting algorithm was proposed. This work contains tree construction modifications as well as new node splitting algorithm. The important feature of this work is using rectangle margin as an optimization criterion of node splitting. This algorithm is similar to Green's algorithm, but has two differences. At first, it chooses axis for splitting that minimizes the sum of margins of MBR groups among all possible sorting-based splits along this axis. At second, it does not split entries at halves, but finds the minimal overlap between all splits based on sorting along this axis. In [12] the comprehensive perfomance analysis of R*-tree is presented. The optimization of R*-tree for non-uniform data is presented in [9].

In [2] a new linear algorithm was proposed. This algorithm makes splits of rectangles along axes based on the closeness of rectangles to value boundaries of the axes. After that, the choice is made among the splits by comparison of the overlaps and distribution ratios.

Since applications of R-tree exist for one-dimensional case, one-dimensional split for R-tree can be considered as a separate problem. One of the negative aspects of R-tree application to one-dimensional case is weak performance of high-overlapping data, such as validity interval or transactional time intervals [11]. This aspect can be partially eliminated by introducing new node splitting algorithm for one-dimensional case which deals better with highly overlapping data.

Guttman's quadratic and linear algorithms can be easily applied to one-dimensional case. For Guttman's quadratic algorithm there is no matter to use quadratic algorithm for picking seeds, because most distant seeds can be found as the intervals which contain the general lower and upper bound, correspondingly. Green's and R*-tree splitting algorithms comprise one-dimensional split as their part. A new linear algorithm also can be applied to one-dimensional case, but we have only one axis for split and will not have to choose among the axes.

III. Proposed algorithm

A. Definitions

In one-dimensional splitting algorithm, the input entries contain a set I of the intervals x_i: $I = \{x_i\}$. An interval is the pair of the lower and the upper bounds: $x_i = (l_i, u_i)$. The general lower bound is $l = min\{l_i\}$, and the general upper bound is $u = max\{u_i\}$. At first, the consideration of splits will be limited by the splits in which one group contains general lower bound and another group contains general upper bound. For this class of splits we will say that a pair (a, b) is a splitting pair, if any interval from I is bounded by (l, a) or (b, u): $\forall x(x \in I \Rightarrow (x \subseteq (l, a)) \wedge (x \subseteq (b, u))$. In other words, a and b are the upper and the lower bound of groups, respectively, for some split of split class under consideration. Let us note that sometimes the splits which are not contained in this class of splits are reasonable. In the figure 3, a split of this class is shown. In the figure 4, a split for the same dataset is shown. In that split, one group stretches from the general lower bound to the general upper bound while another group has rather small area. This split can not be produced by splitting pair.

We will say that the split pair (a, b) is a corner splitting pair if $(a \in \{u_i\}) \wedge (b \in \{l_i\}) \wedge ((\forall t(t < a \Rightarrow \exists x(x \in I \Rightarrow (x \not\subseteq (l, t)) \wedge (x \not\subseteq (b, u)))) \vee (\forall t(t > b \Rightarrow \exists x(x \in I \Rightarrow (x \not\subseteq (l, a)) \wedge (x \not\subseteq (t, u))))))$. In other words, a is one of the upper interval bounds, b is one of the lower interval bounds, and a can not be lower or b can not be higher if the property of being splitting pair still remains. This assumption regarding split seems reasonable since otherwise another split would exist which overlap would be smaller and the minimal number of entries in the group would not be smaller, i.e., there would be a better split in terms of optimization target of this algorithm.

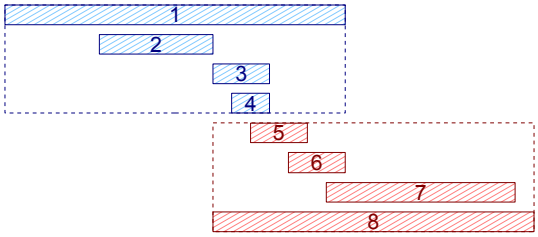

Fig. 3. A split that can be produced by the splitting pair

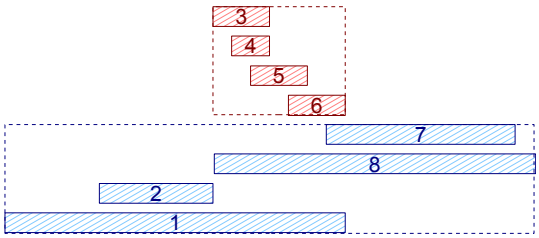

Fig. 4. A split that can not be produced by the splitting pair

B. Algorithm

The algorithm EnumerateCornerSplitPairs(see Algorithm 2) enumerates all corner splitting pairs. The algorithm is based on using two sorted arrays: the first one contains the input entries sorted by the lower bound and the second one contains the input entries sorted by the upper bound. In the main loop of this algorithm, iterations for both arrays are performed simultaneously, so that the property of splitting pair is retained. When a corner splitting pair is found, the ConsiderSplit(see Algorithm 3) is invoked. ConsiderSplit takes the bounding intervals of groups and maximal numbers of entries which can be placed into groups as its input data. Maximal numbers of entries that can be placed into groups are determined using EnumerateCornerSplitPairs by the indexes in the sorted arrays in which the values of splitting pairs are placed. ConsiderSplit reveals the split with minimal overlap of group bounding intervals, where the minimal number of entries in group is greater than or equal to m (m is minimal number of entries in group). When the split with zero overlap is possible, ConsiderSplit chooses the split for which the distance between group bounding intervals is maximal. This property is achieved by allowing the overlap variable to be negative. Let us note that if there are some entries which can be placed into both groups, ConsiderSplit considers the split in which the distribution of entries between groups is closest to the uniform one.

The algorithm DoubleSortSplit(see Algorithm 1) represents the splitting algorithm in general. At first, it invokes EnumerateCornerSplitPairs in order to find allowable corner splitting pair with minimal overlap. Then it distributes entries which can be distributed unambiguously. After that, the rest of entries is sorted by centers of their interval, and they are distributed in a way that makes distribution between groups the most uniform. Since sorting is most time expensibe part of this algorithm, it's time complexity is $O(n \cdot log(n))$ (n – number of input entries).

Algorithm 1 DoubleSortSplit

Input: Overflowed node
Output: Two nodes, at least m entries in each

1: Invoke EnumerateSplitPairs in order to find the corner splitting pair with minimal overlap.
2: Distribute entries that can be placed in only one group into groups.
3: Sort the rest of entries by centers of their intervals.
4: Distribute first m entries to the first group, and distribute other entries to the second group in a way that makes distribution between groups the most uniform.

C. Application to multidimensional case

The proposed algorithm can also be applied to multidimensional case. Algorithm MultidimensionalDoubleSortSplit(see Algorithm 4) represents such an application. At first, it enumerates corner splitting pair along all the axes, and selects the corner splitting pair and the corresponding axis which have the minimal overlap. At second, the entries which can be placed unambiguously are placed. After that the rest of entries are sorted by difference of group area incensement. Finally the split is chosen which has minimal overlap of groups.

IV. Perfomance tests

A. Experimental setup

All the tests were on run on Core 2 Duo 3 GHz computer with 2 GB of memory with Ubuntu 10.10 32bit. For the implementation of R-tree with various node splitting algorithms GiST[8] framework in PostgreSQL DBMS was selected. GiST generalizes various search trees including R-tree.

B. Datasets

Each dataset contains 10^6 randomly generated intervals. The size of intervals conforms to Gaussian distribution with zero mean and the variance that produces the required level of interval overlapping. The level of interval overlapping varied exponentially from 1 to 10^4. The interval center distribution is determined by the dataset type as follows.

- Uniform dataset. The centers of intervals conform to the uniform distribution along interval $[0; 1)$;
- Gaussian dataset. The centers of intervals conform to the standard Gaussian distribution.
- Uniform cluster dataset. At first, 500 cluster centers, which conform to the uniform distribution along interval $[0; 1)$, were generated. After that, for each center 2000 interval centers were generated which offsets from the center conform to the uniform distribution along the interval $[0; 6 * 10^{-4})$.
- Gaussian cluster dataset. At first, 500 cluster centers, which conform to the standard Gaussian distribution, were generated. After that, for each center 2000 interval centers were generated which offsets from the center

Algorithm 2 EnumerateCornerSplitPairs

Input: Set of invervals

Output: Enumeration of all splits that can be produced with corner splitting pairs by invoking ConsiderSplit

Sort intervals by lower bound, write the result to array a
Sort intervals by upper bound, write the result to array b
$s1 \Leftarrow (a[0].l, b[0].u)$
$s2 \Leftarrow (a[0].l, b[n-1].u)$
$i \Leftarrow 0$
$j \Leftarrow 0$
{Iterate until finding a first split produced by the corner splitting pair.}
while $b[j].u = s1.u$ **and** $j < n$ **do**
  $j \Leftarrow j + 1$
end while
considerSplit $(s1, j, s2, n - i)$
while $i < n$ **do**
  $prev\_s2\_l \Leftarrow s2.l$
  $next\_s1\_u \Leftarrow s1.u$
  $next\_i \Leftarrow i$
  {Find next value of s1 upper bound and the corresponding value of s2 lower bound which forms the corner splitting pair with it.}
  while $next\_i < n$ **and** $next\_s2\_l = s2.l$ **do**
    $next\_s1\_u \Leftarrow max\{next\_s1\_u, a[next\_i].u\}$
    $next\_i \Leftarrow next\_i + 1$
    if $next\_i \geq n$ **then**
      break
    end if
    $next\_s2\_l \Leftarrow a[next\_i].l$
  end while
  if $next\_i \geq n$ **and** $next\_s1\_u = s1.u$ **then**
    break
  end if
  {All intermediate values of s2 lower bound form the corner splitting pair with the previous value of s1 upper bound.}
  while $j < n$ **and** $b[j].u \leq next\_s1\_u$ **do**
    if $b[j].u > s1.u$ **and** $b[j].u < next\_s1\_u$ **then**
      $s1.u \Leftarrow b[j]$
      considerSplit $(s1, j + 1, s2, n - i)$
    else
      $s1.u \Leftarrow b[j]$
    end if
    $j \Leftarrow j + 1$
  end while
  {Passage to the next values of s1 upper bound and s2 lower bound.}
  $s1.u \Leftarrow next\_s1\_u$
  $s2.l \Leftarrow next\_s2\_l$
  if $next\_i < n$ **then**
    $i \Leftarrow next\_i$
    considerSplit $(s1, j, s2, n - i)$
  else
    considerSplit $(s1, j, s2, n - i)$
    break
  end if
end while

Algorithm 3 ConsiderSplit

Input: Bounding intervals s1 and s2 of two groups, numbers n1 and n2 which represent the maximal numbers of entries that can be placed into each group.

Output: Updated information regarding the optimal split currently found.

$overlap \Leftarrow (s1.u - s2.l)/(s2.u - s1.l)$
if $n1 \geq m$ **and** $n2 \geq m$ **and** $overlap < best\_overlap$ **then**
  $best\_overlap1 \Leftarrow overlap$
  $best\_s1 \Leftarrow s1$
  $best\_s2 \Leftarrow s2$
  $best\_n1 \Leftarrow n1$
  $best\_n2 \Leftarrow n2$
end if

Algorithm 4 MultidimensionalDoubleSortSplit

Input: Overflowed node

Output: Two nodes, at least m entries in each

Invoke EnumerateSplitPairs for each axis in order to find allowable corner splitting pair with overall minimal overlap.
Distribute entries which can be unambiguously placed into only one group in accordance with the corner splitting pair previously found.
Sort other entries by the difference of group area incensement when adding the entry.
Distribute the first k sorted entries to the first group, and other entries – to the second group, so that the minimal overlap between group MBRs over all possible k is achieved.

conform to the Gaussian distribution with zero mean and the variance of $6 * 10^{-4}$.

For two-dimensional case the datasets were similar. Rather than scalar random values that were generated in the datasets above, vectors of random values having the same distribution that was used in one-dimensional case were generated. Thus these datasets contained rectangles.

C. One-dimensional case

The tests have shown that all sorting-based splitting algorithms perform on this datasets almost equally. This is why only one sorting-based algorithm is represented here, namely, the center sorting algorithm. The following node splitting algorithms were included into tests for one-dimensional case.

- Guttman's quadratic algorithm.
- Center sorting algorithm that searches for the split with minimal level of overlap.
- The proposed double sorting-based algorithm.

In order to compare the efficiency of index structures produced by various splitting algorithms, the numbers of node accesses for query execution were measured. 100 small random intervals having size 10^{-5} were generated for testing, and the

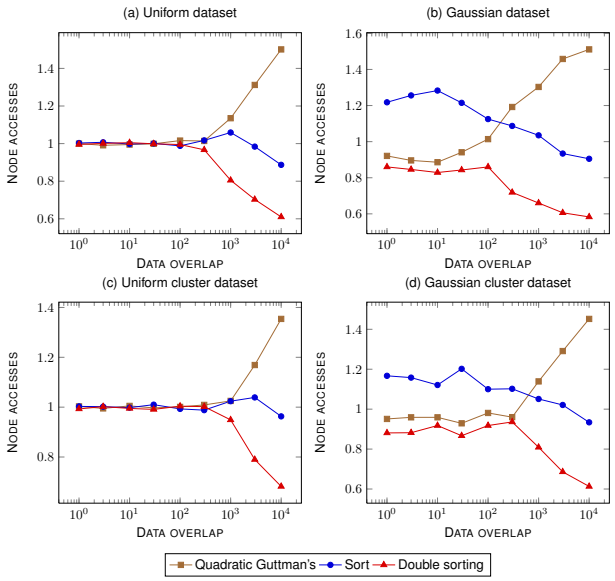

Fig. 5. Comparison of the node access numbers for one-dimensional splitting algorithms

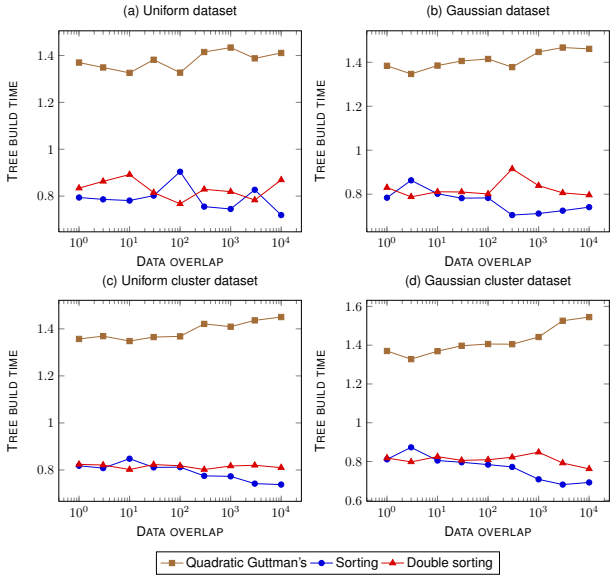

Fig. 6. Comparison of tree building time for one-dimensional splitting algorithms

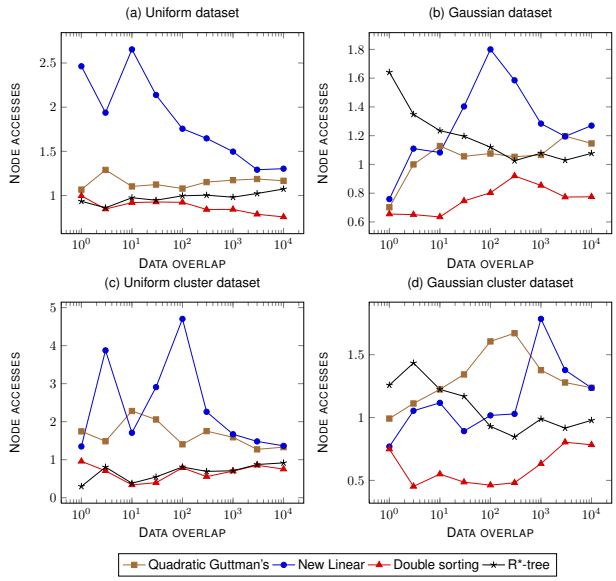

Fig. 7. Comparison of the node access numbers for 2-dimensional splitting algorithms

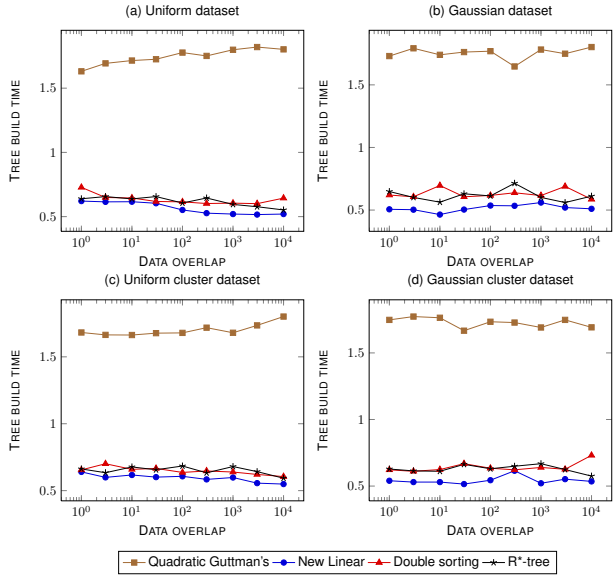

Fig. 8. Comparison of tree building time for 2-dimensional splitting algorithms

number of node accesses required for finding intervals in test datasets that overlap with them was measured. In the figure 5, the average number of node accesses is shown. To simplify the comparison, not absolute value of node access numbers is presented, but rather the ratio of that value for particular algorithm to the average value for all algorithms. The measurements were performed for four datasets described in the subsection above, and for various data overlap levels. In the figure 6 the comparison of tree building times is presented. The data is presented in the same manner as for the data access: as a ratio of building time of particular algorithm to the average building time.

We can see that the number of node accesses required for searching in double sorting-based algorithm is almost never greater than this number in other algorithms. With large data overlap, there is significant superiority of double sorting-based algorithm, up to 50%, in comparison with sorting algorithm, and up to 2 times in comparison with Guttman's quadratic algorithm. We can see that tree construction time for double sorting-based splitting algorithm is smaller than that for Guttman'n quadratic algorithm, but is slightly greater than the time for the sorting algorithm.

D. Two-dimensional case

The following node splitting algorithms were included into tests for two-dimensional case:

- Guttman's quadratic algorithm.
- New linear algorithm.
- Proposed double sorting-based algorithm.
- R*-tree splitting algorithm.

Numbers of node accesses for query execution and tree building time were compared in a same manner as in the one-dimensional case. In the figures7 node access numbers are compared. In the figure 8 tree building times are compared. At first, we can see a weaker correlation between relative node access numbers and the data overlapping. And that correlation is decreased with increasing the number of dimensions. We can see that double sorting-based algorithm shows superiority in terms of node access numbers in most test cases. The tree building time of double sorting-based algorithm is close to that of R*-tree splitting algorithm.

V. Conclusion

In this paper, new double sorting-based node splitting algorithm for R-tree was proposed. This algorithm was initially developed for better handling of complicated cases in one-dimensional split. The proposed splitting algorithm is based on the notion of corner splitting pair and the algorithm of its enumeration. After that, this splitting algorithm was applied to multidimensional cases.

In one-dimensional case, the tests show superiority of the proposed algorithm in terms of the number of node accesses over Guttman's quadratic and simple sorting-based algorithm. The higher superiority was achieved with larger data overlap due to ability of the proposed algorithm to better handle complicated cases. In two-dimensional case, the tests show superiority in terms of number of node accesses over Guttman's quadratic, new linear and R*-tree splitting algorithms in most test cases.

References

[1] A. F. Al-Badarneh, Q. Yaseen, and I. Hmeidi. A new enhancement to the r-tree node splitting. *J. Information Science*, 36(1):3–18, 2010.

[2] C.-H. Ang and T. C. Tan. New linear node splitting algorithm for r-trees. In *Proceedings of the 5th International Symposium on Advances in Spatial Databases*, SSD '97, pages 339–349, London, UK, 1997. Springer-Verlag.

[3] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19:322–331, May 1990.

[5] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30:170–231, June 1998.

[6] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 606–615, Washington, DC, USA, 1989. IEEE Computer Society.

[7] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14:47–57, June 1984.

[8] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[9] K. Kanth, D. Agrawal, A. Singh, and A. E. Abbadi. Indexing non-uniform spatial data. *Database Engineering and Applications Symposium, International*, 0:289, 1997.

[10] C. P. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In J. Clifford and R. King, editors, *SIGMOD Conference*, pages 138–147. ACM Press, 1991.

[11] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31:158–221, June 1999.

[12] Y. Tao and D. Papadias. Performance analysis of r*-trees with arbitrary node extents. *Tran. Knowl. Data Eng. (TKDE*, 16:6–653, 2004.