R^{++}-tree: an efficient spatial access method for highly redundant point data

Martin Šumák, Peter Gurský

P. J. Šafárik University in Košice, Jesenná 5, 04001 Košice, Slovakia
martin.sumak@student.upjs.sk, peter.gursky@upjs.sk

Abstract. We present a new spatial index belonging to R-tree family. Since our new index comes out from the R^+-tree and holds the concept of non-overlapping nodes we call it R^{++}-tree. The original R^+-tree was designed for both point and spatial data. Using R^+-tree for indexing spatial data is very inefficient. In our research we face the problem of indexing product catalogues data that can be represented as point data. Therefore we suggested the R^{++}-tree for point data only. We present a dynamic index R^{++}-tree as an improvement of R^+-tree. In the tests we show that R^{++}-tree offers even better search efficiency than R*-tree when highly redundant point data is considered. Moreover the construction time of R^{++}-tree is much shorter than the construction time of R*-tree.

Keywords: R^+-tree, point data, spatial indexing, spatial searching

1 Introduction

Spatial indexes have been studied for about 30 years. R-trees and its derivatives [1, 2, 3, 4] comprise the most common research branch and are also used in commercial databases. Our motivation for studying R-trees was driven by the need for efficient computation of top-*k* query in product catalogues. This article is not about top-*k* query searching nor top-*k* query evaluation, however it is worth to introduce them at least in a few words.

Imagine we have a large set of apartments for sale and a customer who desires to find an apartment that fits his needs best. For simplification let us reduce customer's criteria to the price and floor. Let the customer want an apartment cheaper than 75 000 € preferably on the second, third or fourth floor. Our mathematical model of top-*k* query would describe such preferences by two functions mapping real values (price, floor number) into values within interval [0; 1] where value 0 indicates not acceptable attribute value and value 1 the most preferred one. These mapping functions are called fuzzy functions and values from [0; 1] are called fuzzy values. Figure 1 shows an example of fuzzy functions of price and floor modeling the preferences of our customer.

For final comparison of any two apartments we use a combination function – weighted sum of fuzzy values, which gives us the overall value of an apartment. The higher is the overall value, the better apartment for the customer is. Let our customer insist on the low price twice as much as on the preferred floor. The combination function C computing the overall value of an apartment A would be: $C(A) = 1 * f_{floor}(v_{floor}(A)) + 2 * f_{price}(v_{price}(A))$ where $v_{floor}(A)$ and $v_{price}(A)$ represent the real values of floor and price of apartment A. Formal description of top-*k* search problem and top-*k* search algorithm can be found in [8]. More relevant comparison of top-*k* search performance over R-tree with other approaches can be found in [9].

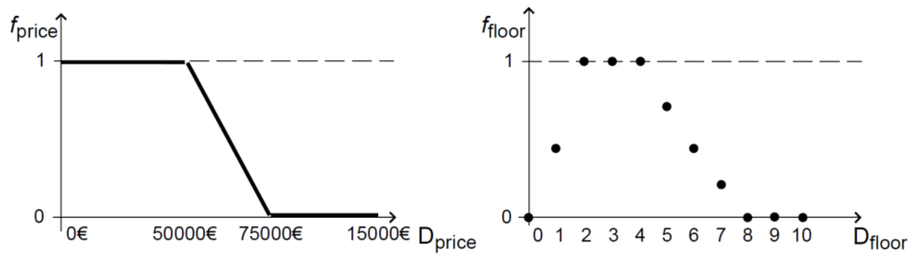


Fig. 1. Customer's preferences to the floor and price of an apartment modeled by fuzzy functions.

It is common in product catalogues, that domains have few possible values of many attributes therefore the redundancy in data is high. Therefore our computational model usually contains top-k query over multidimensional index containing highly redundant point data. Our experiments show that in such scenario R^{++}-tree provides the fastest computation time in comparison to both R-tree and R^*-tree indexes.

In our tests we compare R^{++}-tree with other spatial indexes in top-*k* query, range query and kNN query search. R^{++}-tree is an improved version of R^+-tree [5]. First of all let us remind and conclude the crucial properties of R^+-tree (properties written in italic are all speciality of R^+-tree).

R^+-tree:

- holds nesting condition (as well as R-tree and R*-tree);
- is balanced (as well as R-tree and R*-tree);
- is able to index both point and spatial data (as well as R-tree and R*-tree);
- keeps all data entries in leafs (as well as R-tree and R*-tree);
- each node takes one page of disk space (as well as R-tree and R*-tree) *unless some overflow pages occur;*
- *overflow pages may occur in both leaf and inner nodes;*
- *has no overlaps of nodes at the same level;*
- *leaf node has no guarantees about occupancy;*
- *inner node is guaranteed to have at least 1 entry.*

The R^{++}-tree differs from R^+-tree in following properties:

- it is able to index just point data;
- overflow pages may occur only in leaf nodes;
- each leaf node takes one page of disk space unless some overflow page occurs;
- each inner node takes two disk pages.

All search algorithms (e.g. kNN query, range query and top-*k* query) ideologically work in the same way for all R-trees including R^{++}-tree.

Section 2 summarizes the relevant parts of research papers mentioning R^{+}-tree index. Section 3 describes details of R^{++}-tree design and introduces an algorithm for dynamic insertion. Section 4 reveals conditions (circumstances or data properties) under which the R^{++}-tree offers equal or better search performance than R*-tree. As the title suggests, R^{++}-tree offers the best search performance for highly redundant point data.

2 Related Work

R-tree and R*-tree are the most common indexes from R-tree family. Many tests proved that R*-tree offers the best search performance in most cases. The only area where R*-tree falls behind others is the construction time, because the reinsertions take plenty of time. Special effort was invested to develop an R-tree-like index without overlaps of nodes – the R^{+}-tree. Since this paper is all about our new structure based on R^{+}-tree, we narrow down the related work survey just to two papers describing R^{+}-tree quite thorough. The first one is the original paper, where R^{+}-tree was introduced for the very first time by Timos Sellis et al. [5]. The second one is a paper where R^{+}-tree and three other indexes (R-tree, K-D-B-tree, 2D-Isam) where described and compared in performance of searching and construction by Diane Greene [6].

The main idea of R^{+}-tree presented in [5] is to avoid overlaps between nodes on the same level (strictly just from the structure definition). Condition of zero overlaps leads to several other problems we have to deal with, while executing the dynamic insertion process: (1) finding an appropriate leaf for insertion of a new object, (2) managing overlaps between objects, (3) splitting an overfilled node. Let us discuss each one separately.

(1) The method for finding an appropriate leaf for adding a new object traverses the tree from the root to the leaf along one path. If necessary, the minimal bounding rectangles of nodes are enlarged. Imagine a situation depicted on Figure 2 on the left. No rectangle can be enlarged to cover new object unless an overlap arises. On the right there is an idea of R^{+}-tree depicted – inner node A is completely covered by its child nodes B, C, D, E, F. Therefore the situation depicted on the left never arises and the method always finds an appropriate leaf without any enlargement of the node rectangle. At the beginning of R^{+}-tree construction we have an empty leaf which is simultaneously the root where new objects are added. There is no bounding rectangle of the root stored anywhere. Paper [5] does not discuss explicitly how the bounding rectangles of two new child nodes are supposed to be created. We suppose the bounding rectangles of two new child nodes have to cover all the space where any new data object can appear. We can use some kind of infinite value for rectangle bounds or concrete values, if we know the data scope.

(2) R^{+}-tree is designed for spatial data where data rectangles can overlap. Since rectangles of nodes cannot overlap, R^{+}-tree allows keeping one object concurrently in more leafs. Each involved leaf overlaps only with a part of data rectangle, but all

involved leafs together cover the whole data rectangle. The paper [5] does not discuss a situation, in which there are more multiple overlaps of data rectangles than the leaf capacity is. We suppose that such situation can be handled by creating an overflow page.

(3) Splitting an overfilled node is the most difficult problem. The rectangles of two new nodes cannot overlap with each other and they must cover the rectangle of the original overfilled node. The only solution is to find a cutting hyper-plane and separate child rectangles to the ones falling in front of the hyper-plane and the ones falling behind the hyper-plane. Some child rectangles cut by hyper-plane may occur that cannot be categorized so simply and they must be cut recursively. The distribution of child nodes between the two new nodes is the crucial part of splitting. The split algorithm described in [5] simply says *"pick the next ff rectangles from the list of rectangles sorted on the input axis"*. It offers no technique for searching for an optimal split, e.g. a split causing minimal amount of recursive cuts. Moreover such split may not always be found and different axes and numbers *ff* have to be tried. Therefore we were not able to implement R^+-tree according to description in [5].

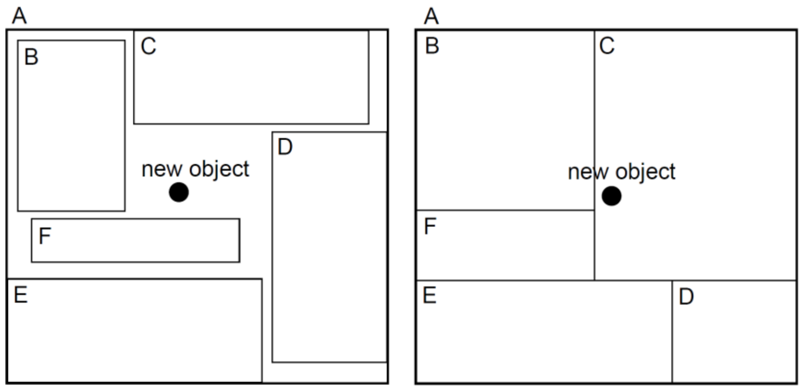


Fig. 2. On the left, no rectangle can be enlarged to cover a new object, unless an overlap with another one arises. On the right inner node A is completely covered by its child nodes B, C, D, E, F, so a new object is added to one of them, in this case to the node C.

Paper [6] brings an analysis and comparison of four spatial access methods: R-tree, K-D-B-tree, R^+-tree and 2D-Isam. There is an algorithm for dynamic insertion into R^+-tree, but split node routine is completely different from the original one proposed in [5], probably because of the same reason why we could not implement it as well. However the split routine proposed in [6] contains a fail branch, exactly by the author's words: "*If still no split exists, R^+-trees (without chaining) fail.*" As the author says, the only way to avoid a fail is to create an overflow page. Using this algorithm an overflow pages can arise for leaf nodes as well as for inner nodes. Split node routine proposed in [6], contrary to the one in [5], is designed to minimize recursive node cuts propagated downwards. Formula, that chooses the split, takes the number of cut child nodes and balance of entries distribution into account.

Both authors conclude that R^{+}-tree is comparable to R-tree for small rectangles without overlaps or point data and it is very inefficient for large rectangles with many overlaps. In product catalogues, products with their attribute values can be represented as multidimensional points and we do not need to index spatial data. These are the main reasons why we restrict our R^{++}-tree for point data use only.

3 R^{++}-Tree

R-tree, R*-tree and R^{+}-tree are designed to store each node on one disk page – each of them with the same fixed size. They all share the same node structure. Leaf entry for an object O is a tuple (*p*(O), *oid*(O)), where *p*(O) is the point of object O and *oid*(O) is an identifier of object O. In other words leaf entry of an object consists of a geometric representation of the object and a pointer to the object possibly residing in external database. Leaf of any of the three trees mentioned above keeps a limited amount of leaf entries. Depending on the implementation, the page keeping a leaf may not keep only the leaf entries, but also a pointer to the parent node and the number of entries present in the leaf. The situation with inner nodes is quite similar. Each inner node keeps a limited amount of inner entries, where each inner entry refers to one child node. Inner entry referring to a child node N is a tuple (*mbr*(N), *nid*(N)), where *mbr*(N) is the minimal bounding rectangle of node N (the geometric representation of node N) and *nid*(N) is an identifier of node N (the pointer to node N). Depending on the implementation, the page keeping an inner node may not keep only the inner entries, but also a pointer to the parent node and the number of entries present in the node.

R-tree, R*-tree and R^{+-}tree differ just in the way of construction. Search algorithms over R^{+}-tree have to handle possible duplicates, since one object can be stored in several leafs. The R^{++}-tree, introduced in this paper, is based on R^{+}-tree, therefore R-tree and R*-tree are left out from further discussion. A thorough description of the dynamic insertion of an object into R^{++}-tree is offered below. Our java implementation of R^{++}-tree can be found at http://ics.upjs.sk/~sumak/files/ rpptree.zip.

3.1 Design of R^{++}-tree

Disadvantage of the original R^{+}-tree is the fact that rectangles of child nodes are rarely minimal. Since rectangle of each node has to be completely covered by rectangles of its child nodes, it is impossible to store minimal bounding rectangles only. The use of minimal bounding rectangles causes troubles when adding new object, as depicted on Figure 2 on the left. On the other hand, larger bounding rectangles, as depicted on Figure 2 on the right, make the search less effective. We propose to keep two rectangles for each child node – the minimal one for searching and the larger one for inserting new objects, see Figure 3. This is the basic idea of R^{++}-tree – to keep an additional rectangle for each child node, which would actually be the minimal bounding rectangle for related child node. In this paper we use the following notation: (1) *br*(N) represents a bounding rectangle of node N but not necessarily the minimal one; (2) *mbr*(N) represents the minimal bounding rectangle of node N; (3) $p(\mathrm{O})$ is the point for object O. The inner node N of R$^+$-tree with parent node P and child nodes M$_1$,..., M$_n$ is: (*nid*(P), *n*, ((*br*(M$_1$), *nid*(M$_1$)),..., (*br*(M$_n$), *nid*(M$_n$)))). The inner node N of R^{++}-tree is: (*nid*(P), *n*, ((*mbr*(M$_1$), *nid*(M$_1$)),..., (*mbr*(M$_n$), *nid*(M$_n$))), (*br*(M$_1$),..., *br*(M$_n$))). The leaf node with parent node P is the same for both R$^+$-tree and R^{++}-tree: (*nid*(P), *n*, ((*p*(O$_1$), *oid*(O$_1$)),..., (*p*(O$_n$), *oid*(O$_n$)))). Figure 3 shows the representation of inner nodes in the pages of size 4096 B.

The structure of R^{++}-tree inner node is designed to keep minimal bounded rectangles together with pointers to child nodes within inner entries. Bounded rectangles are in the second page in the separated list with the same order. Such representation has the following consequences. Each R^{++}-tree inner node takes twice as much space as R$^+$-tree inner node, but when searching, the second page does not have to be read. Since leaf nodes have the same structure in both R$^+$-tree and R^{++}-tree, searching through R^{++}-tree requires reading just one page per node (as it is in R$^+$-tree).

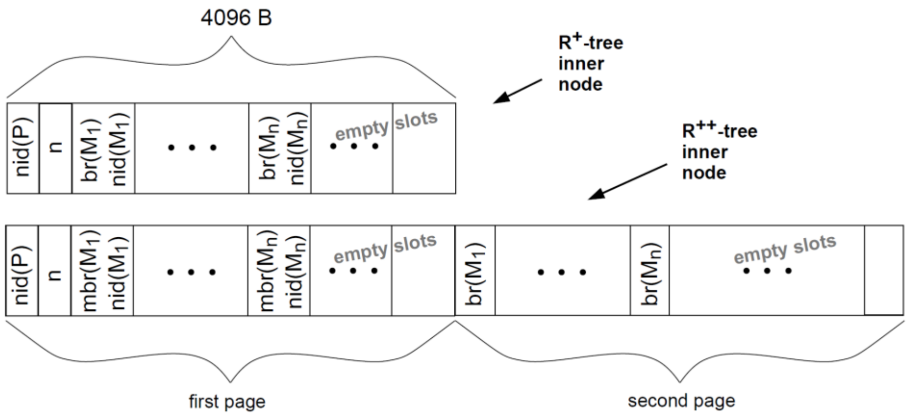


Fig. 3. R^{++}-tree inner node always takes two pages, even in the case of low occupancy, when all data would fit in one page.

Using this approach, the capacity of R^{++}-tree inner node is equal to the capacity of R$^+$-tree inner node with the same page size. The additional information stored in second page has to be read only when adding a new object. Beside the structure of inner node, R^{++}-tree has its own new algorithm for inserting an object. Basically the splitting method is the only new part. Let us remind that we consider point data only. Before describing the algorithm for object insertion itself, we summarize the facts and properties which hold for R^{++}-tree:

1. Leaf node has no occupancy guarantees. Inner node is guaranteed to have at least 1 entry and at least 2 entries if it is the root (node occupancy condition).
2. Bounding rectangle of an inner node completely covers bounding rectangles of its child nodes. Minimal bounding rectangle of an inner node completely covers minimal bounding rectangles of its child nodes. Minimal bounding rectangle of a leaf completely covers points of its objects (nesting condition).
3. Bounding rectangle of an inner node is completely covered by bounding rectangles of its child nodes (complete coverage condition).

4. Bounding rectangle of a node completely covers the minimal bounding rectangle of the node (bounding rectangle vs. minimal bounding rectangle condition).
5. There is no overlap between bounding rectangles of nodes on the same level (zero overlap condition).
6. All leafs are on the same level (balance condition).

3.2 Dynamic Insert

Since all data entries reside in leafs, the first task of inserting a new object is to find an appropriate leaf.

```
Input: node N, object O
Output: leaf L
leaf findLeaf(node N, object O) {
  if N is an inner node {
    Let M be such child node of N, that
    point p(O) falls into rectangle br(M);
    if p(O) does not fall into mbr(M) {
      Enlarge mbr(M) to encompass p(O);
    }
    return findLeaf(M, O);
  }
  if N is a leaf {
    return N;
  }
}
```

If object O lies on the boundary of two rectangles, then arbitrary one is chosen. Such searching goes down the tree along one path and finds one leaf, in which the new object is going to be added. Eventually minimal bounding rectangles along the path are enlarged to encompass the point of a new object. Since complete coverage condition holds true, method **findLeaf** never fails in finding an appropriate child node. Since just the point data is considered, minimal bounding rectangles can be enlarged to cover the new point without violation of any condition.

```
Input: leaf L, object O
Output: -
void insertIntoLeaf(leaf L, object O) {
  Add new leaf entry (p(O), oid(O)) to L;
  if L is overfilled {          //L must be split
    for each dimension d {
      for each entry (p(P), oid(P)) of L {
        Let β be the hyper-plane containing point p(P) and is
        orthogonal to axis of dimension d;
        Compute the difference between number of objects lying
        behind and in front of the hyper-plane β;
      }
    }
    Pick the hyper-plane with minimal difference;
    Move leaf entries of L lying behind the hyper-plane to new
    leaf L';
    if L is the root {
      Create new inner node N and make it the new root;
      insertIntoInner(N, L);
      insertIntoInner(N, L');
    }
    if L is not the root {
      Let N be the parent of L;
      Update mbr(L) and br(L) in inner entry in N according to
      remaining entries in L;
      insertIntoInner(N, L');
    }
  }
}
```

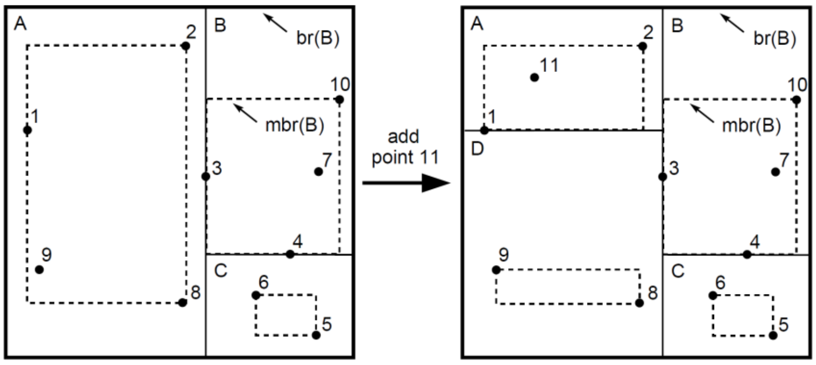


Fig. 4. R^{++}-tree before and after split of leaf node A when adding new object 11. Bounded rectangles are drawn with the full line, the minimal bounded rectangles with the dashed line.

The main issue of inserting process is the splitting of nodes. Splitting a leaf is very simple – the only measure is the balance between number of moved and remained entries. The only problem arises when all objects lie on the same point. In such situation the insert procedure creates an overflow page. In case of many duplicates a chain of overflow pages may occur. Solving such situation by creating a new neighbour leaf and random distribution of entries does not violate the zero overlap condition, because we use point data only. On the other hand it leads to insertion of a new entry into the parent node and possibly to increase of the tree height, which affects search efficiency negatively.

Inserting an object into a leaf may cause an inserting a new entry into its parent inner node. Inserting an entry into an inner node is, if necessary, recursively propagated upwards in the same way.

```
Input: inner node N, node L
Output: -
void insertIntoInner(N, L) {
  Add new inner entry (mbr(L), nid(L)) and br(L) to node N;
  if N is overfilled {                   //N must be split
```

```
    for each dimension d {
      for each entry (mbr(M), nid(M)) of N {
        for both lower and upper bound of mbr(M) in
        dimension d {
          Let β be the hyper-plane containing the bound and
          orthogonal to axis of dimension d;
          Compute the number of rectangles being cut by the
          hyper-plane β;  //measure 1
          Compute the difference between number of rectangles
          not cut by β lying behind and in front of the
          hyper-plane β;  //measure 2
        }
      }
    }
    Pick the hyper-plane cutting minimal number
    of rectangles;  //measure 1
    Resolve ties by picking the one with
    the best balancing;  //measure 2
    Move inner entries (together with relevant additional
    rectangles on second page) lying behind the hyper-plane to
    the new inner node N';
    If there are some rectangles cut by chosen hyper-plane,
    propagate cut downward up to leaf level;
    if N is the root {
      Create new inner node R and make it the new root;
      insertIntoInner(R, N);
      insertIntoInner(R, N');
    }
    if N is not the root {
      Let R be the parent of N;
      Update mbr(N) and br(N) in R according to remaining
      entries in N;
      insertIntoInner(R, N');
    }
  }
}
```

Splitting an inner node is more difficult than splitting a leaf. We evaluate two measures for each tangent hyper-plane to side of a hyper-rectangle of a child node. First of all we try to prevent cuts of child nodes, but it is not always possible to avoid them. Due to the zero overlap condition the cut has to be propagated downward. That may cause a non-optimal cutting of nodes on lower levels. That leads to nodes crumbling, which affects the searching efficiency negatively. Contrary to the leaf nodes, the cut of inner node is always guaranteed to be found and no overflow pages are necessary. To prove this claim, we have to prove, that each time an inner node is overfilled, there is a hyper-plane with at least one child rectangle lying behind and at least one child rectangle lying in front of this hyper-plane. Since splitting of an inner node always comes after a splitting of a child node, the overfilled inner node contains the original child and the new neighbour child created by the split. Due to this fact the hyper-plane used for splitting the child can be used for splitting its parent, because it is guaranteed that one part of the original child lies behind and the other one in front of the hyper-plane.

3.3 Special Issues of Dynamic Insert

There are two more issues not discussed explicitly in the pseudo-code of insertion process. The first one is how to determine boundaries of two nodes arisen from the root split. The second issue is about possibly empty leaf nodes.

Let us look at the first issue. After splitting root node the minimal bounding rectangles of its child nodes are easy to compute. The minimal bounding rectangles cannot be used as bounding rectangles residing on the second page of node, unless they together completely cover all domain in each dimension. Since no enlargements of bounding rectangles are allowed in method `findLeaf` (note that only minimal bounding rectangles are enlarged if necessary) the root's children have to completely cover the whole space, where data can appear. We can concrete values when we know the limits of data or we can use infinity values.

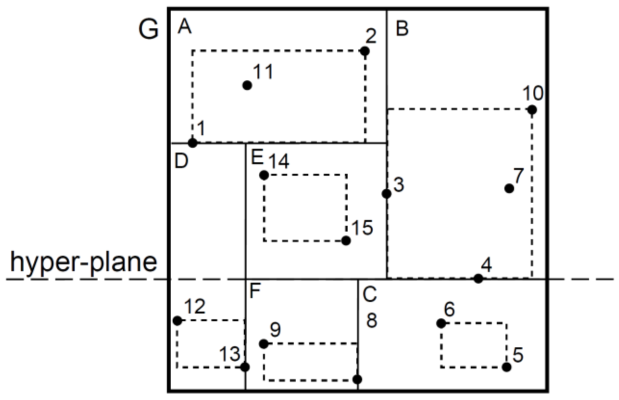


Fig. 5. Leaf node D is to be cut by hyper-plane and its part above the hyper-plane is empty.

The second issue not discussed explicitly, is empty nodes. Imagine a situation on Figure 5 (splitting of node G), where leaf D is forced to be cut by a hyper-plane and all of its data fall in front of the hyper-plane. The rest of the leaf D behind the hyper-plane is empty and we cannot determine the minimal bounding rectangle of the new leaf. Since we cannot violate the complete coverage condition, we cannot leave out this node of the tree. We propose to keep this node with the information, that it is empty. Inner node is not designed to keep any information about number of entries residing in its child nodes. Therefore, we propose to use some kind of a *null* value for the minimal bounding rectangle to determine that related child node is empty.

We leave out the description of search algorithms because all of them (for range queries, kNN queries, top-*k* queries and others) work exactly the same way they work in R-tree or R*-tree. The only important thing is to read just first pages of inner nodes to use minimal bounding rectangles. However, using bounding rectangles from the second page, i.e. not the minimal ones, does not cause an incorrect search computation, it simply leads to lower search performance comparable to original R^+-tree.

4 Experiments

Since the proposed R^{++}-tree is designed for point data, we used several sets of synthetic point data and pseudo-real point data in the tests. We compared R^{++}-tree with R-tree and R*-tree. The main measure was the search time efficiency of range query, kNN query and top-*k* query. We used 4 kB pages for all the tests because it is the size of allocation unit on disks.

Comprehensive tests require extremely big effort due to many variables in tests that need to be fixed. Point data can vary in dimensionality, distribution, density and redundancy. Range queries can vary in the area of range, dimensionality and kNN queries can vary in the number *k* and location of reference point. Top-*k* queries can vary in number *k*, weights and fuzzy functions.

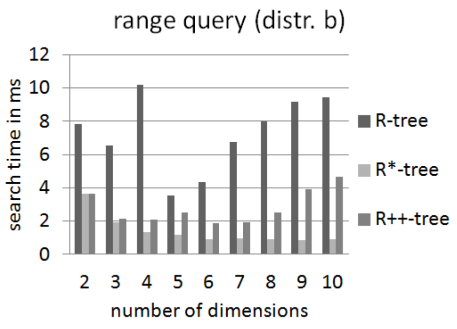
In the tests we used the following data distributions: (a, b, c) – synthetic data, (d) – pseudo-real data. Distribution (a) consists of uniformly distributed random points within interval [0; 1] in each dimension and with precision of coordinates to fifteen decimal places. Distributions (b), (c) consist of uniformly distributed random points with integer coordinates within interval [0; 100], [0; 10] in each dimension respectively. Distribution (d) is based on real data set containing approximately 27 000 flat or house advertisements in Slovakia having 6 attributes: price, area, floor, the highest floor of building, year of approbation and the number of rooms. Values in all 6 attributes are numbers, so we can easily represent each flat by a point in 6-dimensional space. Since the real data set was small, we generated bigger pseudo-real sets by generation of several similar objects for each one from the original set. This way we generated two sets, one with about 550 000 objects (the 20-multiple set) and one with about 2 700 000 objects (the 100-multiple set).

4.1 Synthetic data

Let us discuss the experiments over pure synthetic data (i.e. distributions a, b, c). For each distribution we generated 100 000 random points with dimensionality from 2 to 10 dimensions, i.e. 27 sets of data altogether. For each set of data we built three types of trees (R-tree, R*-tree and R^{++}-tree) i.e. 81 trees. We prepared 300 random queries for each data set and each query type. We generated ranges of 3 size types with uniformly random position for range queries. For each kNN query we generated reference point with uniformly random position and number *k* uniformly random from 1 to 100. Similarly we generated random top-*k* queries containing random weights from 1 to 5, random numbers *k* from 1 to 100 and reasonable fuzzy functions.

Distribution (a) has almost no redundancy and minimal bounding rectangles of R^{++}-tree are almost the same size as the bounding rectangles. The result is that R^{++}-tree is very inefficient for all query types. However these data are quite different from real product catalogues data. We do not provide any results in this paper.

Distribution (b) has many redundancies in low dimensional spaces and just a few redundancies in higher dimensional spaces. Search performance of R^{++}-tree is comparable to R*-tree in all query types for low dimensional spaces. However, even for high dimensional spaces, R^{++}-tree is almost always better than R-tree (Figures 6 and 7).



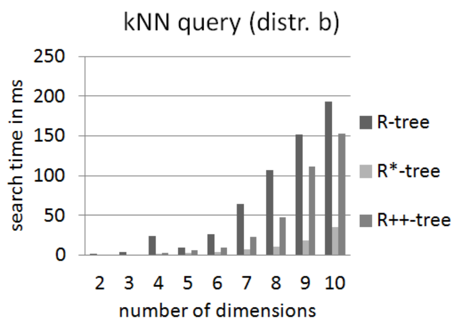
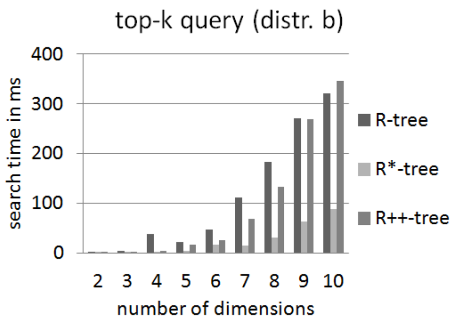


Fig. 6. Average search time (in milliseconds) per range query and kNN query over data with distribution (b).



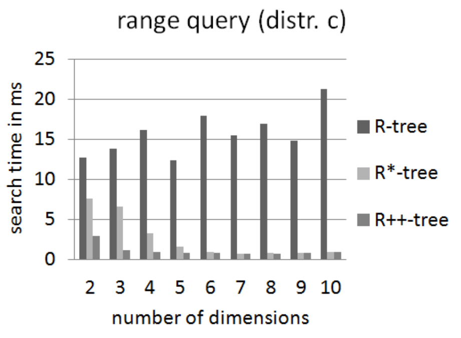
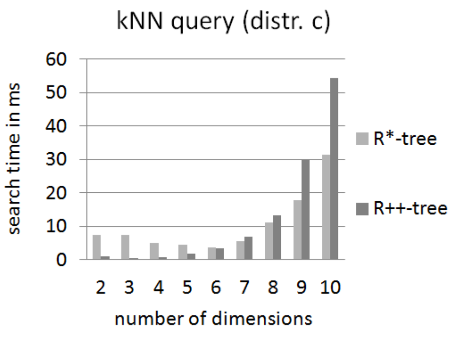


Fig. 7. Average search time (in milliseconds) per top-k query over data with distribution (b) and range query over data with distribution (c).



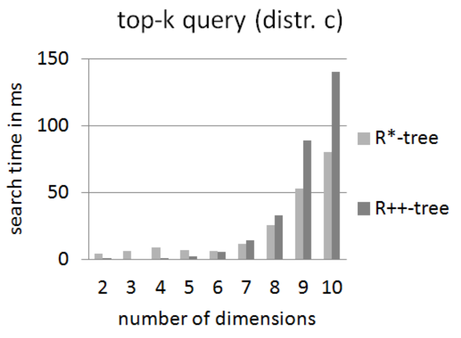


Fig. 8. Average search time (in milliseconds) per kNN query and top-k query over data with distribution (c).

Distribution (c) contains many redundancies in low dimensional spaces. In this case R^{++}-tree is the best one. Even in higher dimensional spaces (from 5 to 10 dimensions) R^{++}-tree is comparable to R*-tree especially in time of range query search. We left out the R-tree of the charts on Figure 8 because its significantly worse results make the differences between R*-tree and R^{++}-tree illegible.

R^{++}-tree is significantly more efficient than R*-tree in all queries up to 4 dimensional space. In more dimensional spaces there are less redundancies and R*-tree becomes again the most efficient one.

4.2 Pseudo-real data

As mentioned earlier we used two 6-dimensional sets of pseudo-real data: the 20-multiple set (550 000 objects) and the 100-multiple set (2 700 000 objects). We compared the average time of top-*k* query search over R-tree, R*-tree and R^{++}-tree. Each of the tests consists of the same set of 1100 random top-*k* queries, i.e. about 200 random queries containing gradually 2, 3, 4, 5 and all 6 attributes. In other words not all queries contained all 6 attributes and not all *n*-attribute queries contained the same attributes. All queries for the 20-multiple set have number k = 25 and all queries for the 100-multiple set have number k = 50.

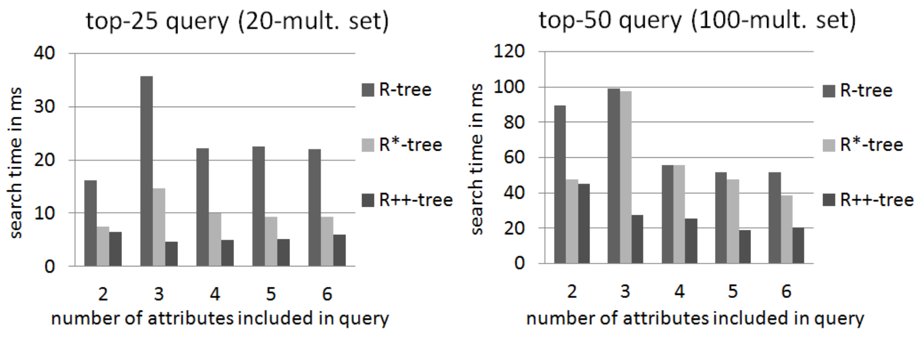


Fig. 9. Time of top-25 query over 20-multiple and top-50 query over 100-multiple data set.

As we can see on Figure 9, R^{++}-tree offers better top-*k* search performance than R*-tree and much better top-*k* search performance than R-tree. We suppose it is the result of many redundancies in real data, caused especially by attributes: number of rooms, floor, highest floor of building, year of approbation, which have few possible values.

Note that axis *x* contains the number of attributes used in top-*k* query, not the space dimensionality. We emphasize that all the trees contain full 6-dimensional data. On the charts we see another interesting property – top-*k* query performance seems to be invariant to the number of dimensions included in the query for all types of R-tree (including R^{++}-tree).

In our tests we compared R^{++}-tree with R-tree and R*-tree as the most common indexes from R-tree family. We omitted the R^{+}-tree (its implementation according to [6]) from the tests because we found it to be really inefficient.

Conclusion

We present a new R-tree like index – the R^{++}-tree as an improvement of R^{+}-tree. Even if R*-tree seems to be universal index and the best in search time in many cases, it falls behind in the efficiency of insertion process. We found out that R^{++}-tree is significantly more efficient than R*-tree for range query, kNN query and top-*k* query, when point data with many redundancies is considered. Test results with synthetic data for distribution (c) show that R^{++}-tree is the best up to 4 dimensions. The efficiency of R^{++}-tree slightly decreases with growing dimensionality, because the number of redundancies decreases too. Tests over pseudo-real data also showed that R^{++}-tree offers very good search performance for top-*k* query, which is the main motivation for our research.

References

1. Guttman, A.: A dynamic index structure for spatial searching. SIGMOD Conference. (1997)
2. Theodoridis, Y., Sellis, T.: Optimization Issues in R-tree Construction. In Proceedings of the International Workshop on Geographic Information Systems. (1993)
3. Brakatsoulas, S., Pfoser, D., Theodoridis, Y.: Revisiting R-tree Construction Principles. Proceedings of ADBIS the 6th East European Conference on Advances in Databases and Information. (2002) 149–162.
4. Beckmann, N., Kriegel, H. P., Schneider, R., Seeger, B.: The R*-Tree: An efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference. (1990) 322–331
5. Sellis, T., Roussopoulos, N., Faloutsos, C.: The R+-Tree: A dynamic index for multi-dimensional objects. In VLDB. (1987)
6. Greene, D.: An Implementation and Performance Analysis of Spatial Data Access Methods. Proc. Fifth Int'l Conf. Data Eng. (1989) 606–615.
7. Hjaltason, G. R., Samet, H.: Distance browsing in spatial databases. ACM Transactions on Database Systems. (1999) 265–318
8. Šumák, M., Gurský, P.: Top-k search in product catalogues. Proceedings of DATESO. (2011) 1 – 12
9. Šumák, M., Gurský, P.: Top-k search over grid file. Proceedings of DATESO. (2012) 115–126