1. Introduction

The paper entitled "The ubiquitous B-tree" by Comer was published in *ACM Computing Surveys* in 1979 [49]. Actually, the keyword "B-tree" was standing as a generic term for a whole family of variations, namely the B^*-tree, the B^+-tree and several other variants [111]. The title of the paper might have seemed provocative at that time. However, it represented a big truth, which is still valid now, because all textbooks on databases or data structures devote a considerable number of pages to explain definitions, characteristics, and basic procedures for searches, inserts, and deletes on B-trees. Moreover, B^+-trees are not just a theoretical notion. On the contrary, for years they have been the de facto standard access method in all prototype and commercial relational systems for typical transaction processing applications, although one could observe that some quite more elegant and efficient structures have appeared in the literature.

The 1980s were a period of wide acceptance of relational systems in the market, but at the same time it became apparent that the relational model was not adequate to host new emerging applications. Multimedia, CAD/CAM, geographical, medical and scientific applications are just some examples, in which the relational model had been proven to behave poorly. Thus, the object-oriented model and the object-relational model were proposed in the sequel. One of the reasons for the shortcoming of the relational systems was their inability to handle the new kinds of data with B-trees. More specifically, B-trees were designed to handle alphanumeric (i.e., one-dimensional) data, like integers, characters, and strings, where an ordering relation can be defined. A number of new B-tree variations have appeared in the literature to handle object-oriented data (see [25] for a comparative study). Mainly, these structures were aimed at hosting data of object hierarchies in a single structure. However, these efforts had limited applicability and could not cover the requirements of many new application areas.

In light of this evolution, entirely novel access methods were proposed, evaluated, compared, and established. One of these structures, the R-tree, was proposed by Guttman in 1984, aimed at handling geometrical data, such as points, line segments, surfaces, volumes, and hypervolumes in high-dimensional spaces [81]. R-trees were treated in the literature in much the same way as B-trees. In particular, many improving variations have been proposed for various

instances and environments, several novel operations have been developed for them, and new cost models have been suggested.

It seems that due to modern demanding applications and after academia has paved the way, the industry recently recognized the use and necessity of R-trees. Thus, R-trees are adopted as an additional access method to handle multi-dimensional data. Based on the observation that "trees have grown everywhere" [212], we anticipate that we are in the beginning of the era of the "ubiquitous R-tree" in an analogous manner as B-trees were considered 25 years ago. Nowadays, spatial databases and geographical information systems have been established as a mature field, spatiotemporal databases and manipulation of moving points and trajectories are being studied extensively, and finally image and multimedia databases able to handle new kinds of data, such as images, voice, music, or video, are being designed and developed. An application in all these cases should rely on R-trees as a necessary tool for data storage and retrieval. R-tree applications cover a wide spectrum, from spatial and temporal to image and video (multimedia) databases. The initial application that motivated Guttman in his pioneering research was VLSI design (i.e., how to efficiently answer whether a space is already covered by a chip). Gradually, handling rectangles quickly found applications in geographical and, in general, spatial data, including GIS (buildings, rivers, cities, etc.), image or video/audio retrieval systems (similarity of objects in either original space or high-dimensional feature space), time series and chronological databases (time intervals are just 1D objects), and so on. Therefore, we argue that R-trees are found everywhere.

We begin the exploration of the R-tree world with Table 1.1, which shows all R-tree variations covered in this book. For each R-tree variation we give the author(s), the year of publication, and the corresponding reference number. In Table 1.2 we give the most important symbols and the corresponding descriptions used throughout the book. The next section presents the structure and basic characteristics of the original R-tree access method proposed in [81].

Table 1.1. Access methods covered in this book, ordered by year of publication.

Year	Access Method	Authors and References
1984	R-tree	Guttman [81]
1985	Packed R-tree	Roussopoulos, Leifker [199]
1987	R^+-tree	Sellis, Roussopoloulos, Faloutsos [211]
1989	Cell-tree	Guenther [77]
1990	P-tree	Jagadish, [96] (and 1993 Schiwietz [206])
1990	R^*-tree	Beckmann, Kriegel, Schneider, Seeger [19]
1990	RT-tree	Xu, Han, Lu [249]
1990	Sphere-tree	Oosterom [164]
1992	Independent R-trees	Kamel, Faloutsos [103]
1992	MX R-tree	Kamel, Faloutsos [103]
1992	Supernode R-tree	Kamel, Faloutsos [103]
1993	Hilbert Packed R-tree	Kamel, Faloutsos [104]

Table 1.1. Access methods covered in this book, ordered by year of publication (continued).

Year	Access Method	Authors and References
1994	Hilbert R-tree	Kamel, Faloutsos [105]
1994	R-link	Ng, Kameda [161]
1994	TV-tree	Lin, Jagadish, Faloutsos [138]
1996	QR-tree	Manolopoulos, Nardelli, Papadopoulos, Proietti [146]
1996	SS-tree	White, Jain [245]
1996	VAMSplit R-tree	White, Jain [244]
1996	X-tree	Berchtold, Keim, Kriegel [24]
1996	3D R-tree	Theodoridis, Vazirgiannis, Sellis [238]
1997	Cubtree	Roussopoulos, Kotidis [198]
1997	Linear Node Splitting	Ang, Tan [11]
1997	S-tree	Aggrawal, Wolf, Wu, Epelman [5]
1997	SR-tree	Katayama, Satoh [108]
1997	STR R-tree	Leutenegger, Edgington, Lopez [134]
1998	Bitemporal R-tree	Kumar, Tsotras, Faloutsos [125]
1998	HR-tree	Nascimento, Silva [158, 159]
1998	Optimal Node Splitting	Garcia, Lopez, Leutenegger [71]
1998	R^*_a-tree	Juergens, Lenz [102]
1998	STLT	Chen, Choubey, Rundensteiner [42]
1998	TGS	Garcia, Lopez, Leutenegger [70]
1999	GBI	Choubey, Chen, Rundensteiner [47]
1999	R^{ST}-tree	Saltenis, Jensen [201]
1999	2+3 R-tree	Nascimento, Silva, Theodoridis [159]
2000	Branch Grafting	Schrek, Chen [208]
2000	Bitmap R-tree	Ang, Tan [12]
2000	TB-tree	Pfoser, Jensen, Theodoridis [189]
2000	TPR-tree	Saltenis, Jensen, Leutenegger, Lopez [202]
2001	aR-tree	Papadias, Kanlis, Zhang, Tao [170]
2001	Box-tree	Agarwal, deBerg, Gudmundsson, Hammar, Haverkort [4]
2001	Compact R-tree	Huang, Lin, Lin [93]
2001	CR-tree	Kim, Cha, Kwon [110]
2001	Efficient HR-tree	Tao, Papadias [222]
2001	MV3R-tree	Tao, Papadias [223]
2001	PPR-tree	Kollios, Tsotras, Gunopulos, Delis, Hadjieleftheriou [113]
2001	RS-tree	Park, Heu, Kim [184]
2001	SOM-based R-tree	Oh, Feng, Kaneko, Makinouchi [162]
2001	STAR-tree	Procopiuc, Agarwal, Har-Peled, [192]
2002	aP-tree	Tao, Papadias, Zhang, [228]
2002	Buffer R-tree	Arge, Hinrichs, Vahrenhold, Vitter, [16]
2002	cR-tree	Brakatsoulas, Pfoser, Theodoridis, [32]
2002	DR-tree	Lee, Chung, [133]
2002	HMM R-tree	Jin, Jagadish, [100]
2002	Lazy Update R-tree	Kwon, Lee, Lee, [127]
2002	Low Stabbing Number	deBerg, Hammar, Overmars, Gudmundsson, [56]
2002	VCI R-tree	Prabhakar, Xia, Kalashnikov, Aref, Hambrusch, [191]
2003	FNR-tree	Frentzos, [67]
2003	LR-tree	Bozanis, Nanopoulos, Manolopoulos, [31]
2003	OMT R-tree	Lee, Lee, [131]
2003	Partitioned R-tree	Bozanis, Nanopoulos, Manolopoulos, [31]
2003	Q+R-tree	Xia, Prabhakar, [248]
2003	Seeded Clustering	Lee, Moon, Lee, [132]
2003	SETI	Chakka, Everspaugh, Patel, [38]
2003	TPR^*-tree	Tao, Papadias, Sun, [227]
2003	TR-tree	Park, Lee, [185]
2004	Merging R-trees	Vasatitis, Nanopoulos, Bozanis, [240]
2004	MON-tree	Almeida, Guting, [7]
2004	PR-tree	Arge, deBerg, Haverkort, Yi, [15]
2004	R^{PPF}-tree	Pelanis, Saltenis, Jensen, [188]
2004	VMAT	Gorawski, Malczok, [73, 74]

Table 1.2. Basic notation used throughout the study, listed in alphabetical order.

Symbol	Description
$\mathcal{B}$	set of buckets
B_i	a bucket
b	bucket capacity in bytes
c	R-tree leaf node capacity
C_{MJJ}	cost of a multi-way spatial join query
C_{NN}	cost of a nearest-neighbor query
C_{SJ}	cost of a pair-wise join query
C_W	cost of a window query
Den	density of dataset
d	dataset dimensionality
$\mathcal{E}$	set of node entries
e, E	R-tree node entry
$e.mbr, E.mbr$	R-tree node entry MBR
f	R-tree fanout (non-leaf node capacity)
FD_0	Hausdorff fractal dimension
FD_2	correlation fractal dimension
H	Hilbert value
h	R-tree height
k	number of nearest neighbors
L	R-tree leaf node
M	maximum number of entries in an R-tree node
m	minimum number of entries in an R-tree node
N	number of data objects (dataset cardinality)
n	total number of nodes
n_l	number of leaf nodes
o, O	data object
$o.mbr, O.mbr$	object minimum bounding rectangle (MBR)
oid	object identifier
ptr	pointer to a node
q, Q	query object (point/rectangle/polygon)
$q.mbr, Q.mbr$	query object MBR
r	data object (point/rectangle/polygon)
RN	R-tree node
$RN.mbr$	R-tree node MBR
RN_l	R-tree leaf node
$RN.type$	type of node (leaf or internal)
$\mathcal{RS}$	set of data rectangles
σ	selectivity of a spatial query
$\sigma(k)$	index selectivity for k-CP query
T	a tree
t_{end}	interval ending time
t_{start}	interval starting time

1.1 The Original R-tree

Although, nowadays the original R-tree [81] is being described in many standard textbooks and monographs on databases [130, 147, 203, 204], we briefly recall its basic properties. R-trees are hierarchical data structures based on B^+-trees. They are used for the dynamic organization of a set of d-dimensional geometric objects representing them by the minimum bounding d-dimensional rectangles (for simplicity, MBRs in the sequel). Each node of the R-tree corresponds to the MBR that bounds its children. The leaves of the tree contain pointers to the database objects instead of pointers to children nodes. The nodes are implemented as disk pages.

It must be noted that the MBRs that surround different nodes may overlap each other. Besides, an MBR can be included (in the geometrical sense) in many nodes, but it can be associated to only one of them. This means that a spatial search may visit many nodes before confirming the existence of a given MBR. Also, it is easy to see that the representation of geometric objects through their MBRs may result in false alarms. To resolve false alarms, the candidate objects must be examined. For instance, Figure 1.1 illustrates the case where two polygons do not intersect each other, but their MBRs do. Therefore, the R-tree plays the role of a filtering mechanism to reduce the costly direct examination of geometric objects.

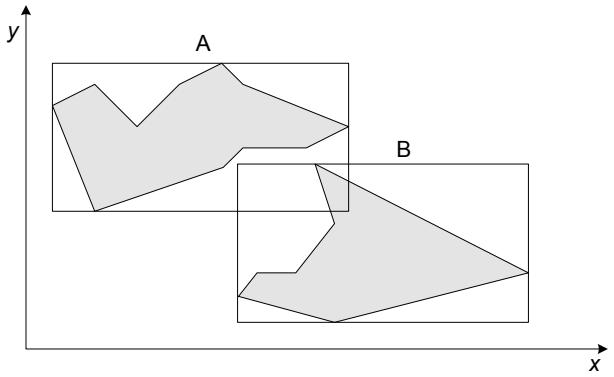

Fig. 1.1. An example of intersecting MBRs, where the polygons do not intersect.

An R-tree of order (m, M) has the following characteristics:

- Each leaf node (unless it is the root) can host up to M entries, whereas the minimum allowed number of entries is $m \leq M/2$. Each entry is of the form (mbr, oid), such that mbr is the MBR that spatially contains the object and oid is the object's identifier.
- The number of entries that each internal node can store is again between $m \leq M/2$ and M. Each entry is of the form (mbr, p), where p is a pointer to a child of the node and mbr is the MBR that spatially contains the MBRs contained in this child.

– The minimum allowed number of entries in the root node is 2, unless it is a leaf (in this case, it may contain zero or a single entry).
– All leaves of the R-tree are at the same level.

From the definition of the R-tree, it follows that it is a height-balanced tree. As mentioned, it comprises a generalization of the B^+-tree structure for many dimensions. R-trees are dynamic data structures, i.e., global reorganization is not required to handle insertions or deletions.

Figure 1.2 shows a set of the MBRs of some data geometric objects (not shown). These MBRs are $D, E, F, G, H, I, J, K, L, M$, and N, which will be stored at the leaf level of the R-tree. The same figure demonstrates the three MBRs (A, B, and C) that organize the aforementioned rectangles into an internal node of the R-tree. Assuming that $M = 4$ and $m = 2$, Figure 1.3 depicts the corresponding MBR. It is evident that several R-trees can represent the same set of data rectangles. Each time, the resulting R-tree is determined by the insertion (and/or deletion) order of its entries.

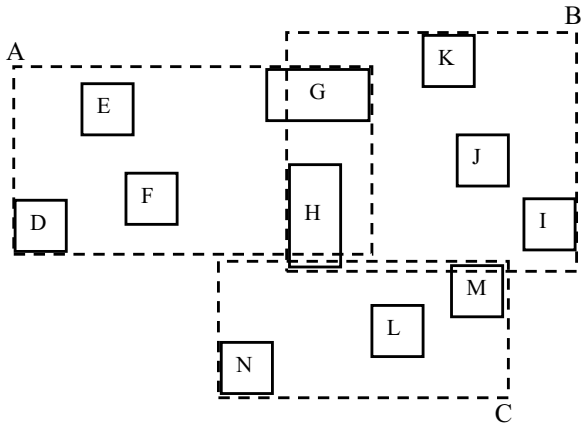

Fig. 1.2. An example of data MBRs and their MBRs.

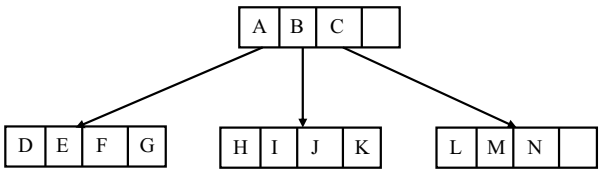

Fig. 1.3. The corresponding R-tree.

Let an R-tree store N data rectangles. In this case the maximum value for its height h is:

$$h_{\max} = \lceil \log_m N \rceil - 1 \tag{1.1}$$

The maximum number of nodes can be derived by summing the maximum possible number of nodes per level. This number comes up when all nodes contain the minimum allowed number of entries, i.e., m. Therefore, it results that the maximum number of nodes in an R-tree is equal to:

$$\sum_{i=1}^{h_{\max}} \lceil N/m^i \rceil = \lceil N/m \rceil + \lceil N/m^2 \rceil + \ldots + 1$$

Given a rectangle, Q, we can form the following query: find all data rectangles that are intersected by Q. This is denoted as a range (or window) query. The algorithm that processes range queries in an R-tree is given in Figure 1.4. For a node entry $E, E.mbr$ denotes the corresponding MBR and $E.p$ the corresponding pointer to the next level. If the node is a leaf, then $E.p$ denotes the corresponding object identifier (*oid*).

Algorithm RangeSearch(TypeNode RN**, TypeRegion** Q**)**
/* Finds all rectangles that are stored in an R-tree with root node RN, which are intersected by a query rectangle Q. Answers are stored in the set $\mathcal{A}$ */

1. **if** RN is not a leaf node
2. examine each entry e of RN to find those $e.mbr$ that intersect Q
3. **foreach** such entry e call RangeSearch($e.ptr$,Q)
4. **else** // RN is a leaf node
5. examine all entries e and find those for which $e.mbr$ intersects Q
6. add these entries to the answer set $\mathcal{A}$
7. **endif**

Fig. 1.4. The R-tree range search algorithm.

We note that the rectangles that are found by range searching constitute the candidates of the filtering step. The actual geometric objects intersected by the query rectangle have to be found in a refinement step by retrieving the objects of the candidate rectangles and testing their intersection.

Insertions in an R-tree are handled similarly to insertions in a B^+-tree. In particular, the R-tree is traversed to locate an appropriate leaf to accommodate the new entry. The entry is inserted in the found leaf and, then all nodes within the path from the root to that leaf are updated accordingly. In case the found leaf cannot accommodate the new entry because it is full (it already contains M entries), then it is split into two nodes. Splitting in R-trees is different from that of the B^+- tree, because it considers different criteria. The algorithm for inserting a new data rectangle in an R-tree is presented in Figure 1.5.

The aforementioned insertion algorithm uses the so-called *linear split* algorithm (it has linear time complexity). The objective of a split algorithm is to minimize the probability of invoking both created nodes (L_1 and L_2) for

Algorithm Insert(TypeEntry E, TypeNode RN)
/* Inserts a new entry E in an R-tree with root node RN */

1. Traverse the tree from root RN to the appropriate leaf. At each level, select the node, L, whose MBR will require the minimum area enlargement to cover $E.mbr$
2. In case of ties, select the node whose MBR has the minimum area
3. **if** the selected leaf L can accommodate E
4. Insert E into L
5. Update all MBRs in the path from the root to L, so that all of them cover $E.mbr$
6. **else** // L is already full
7. Let $\mathcal{E}$ be the set consisting of all L's entries and the new entry E Select as seeds two entries $e_1, e_2 \in \mathcal{E}$, where the distance between e_1 and e_2 is the maximum among all other pairs of entries from $\mathcal{E}$ Form two nodes, L_1 and L_2, where the first contains e_1 and the second e_2
8. Examine the remaining members of $\mathcal{E}$ one by one and assign them to L_1 or L_2, depending on which of the MBRs of these nodes will require the minimum area enlargement so as to cover this entry
9. **if** a tie occurs
10. Assign the entry to the node whose MBR has the smaller area
11. **endif**
12. **if** a tie occurs again
13. Assign the entry to the node that contains the smaller number of entries
14. **endif**
15. **if** during the assignment of entries, there remain λ entries to be assigned and the one node contains $m - \lambda$ entries
16. Assign all the remaining entries to this node without considering the aforementioned criteria /* so that the node will contain at least m entries */
17. **endif**
18. Update the MBRs of nodes that are in the path from root to L, so as to cover L_1 and accommodate L_2
19. Perform splits at the upper levels if necessary
20. In case the root has to be split, create a new root
21. Increase the height of the tree by one
22. **endif**

Fig. 1.5. The R-tree insertion algorithm.

the same query. The linear split algorithm tries to achieve this objective by minimizing the total area of the two created nodes. Examples of bad and good splits are given in Figure 1.6. In the left part of the figure, the split is bad, because the MBRs of the resulting nodes have much larger area than that depicted in the right part of the figure.

The linear split algorithm, however, is one of the three alternatives to handle splits that were proposed by Guttman. The other two are of quadratic and exponential complexity. These three alternatives are summarized as follows:

Linear Split. Choose two objects as seeds for the two nodes, where these objects are as far apart as possible. Then consider each remaining object in a

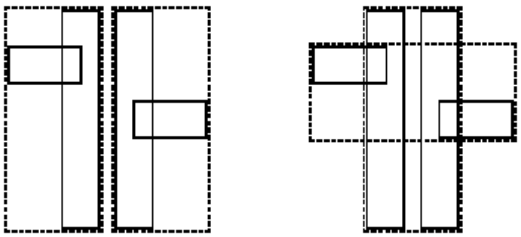

Fig. 1.6. Left: bad split; Right: good split.

random order and assign it to the node requiring the smallest enlargement of its respective MBR.

Quadratic Split. Choose two objects as seeds for the two nodes, where these objects if put together create as much dead space as possible (*dead space* is the space that remains from the MBR if the areas of the two objects are ignored). Then, until there are no remaining objects, insert the object for which the difference of dead space if assigned to each of the two nodes is maximized in the node that requires less enlargement of its respective MBR.

Exponential Split. All possible groupings are exhaustively tested and the best is chosen with respect to the minimization of the MBR enlargement.

Guttman suggested using the quadratic algorithm as a good compromise to achieve reasonable retrieval performance.

Algorithm Delete(TypeEntry E**, TypeNode** RN**)**
/* Deletes an entry E from an R-tree with root node RN */

1. **if** RN is a leaf node
2. search all entries of RN to find $E.mbr$
3. **else** // RN is an internal node
4. Find all entries of RN that cover $E.mbr$
5. Follow the corresponding subtrees until the leaf L that contains E is found
6. Remove E from L
7. **endif**
8. Call algorithm **CondenseTree(**L**)** /* Figure 1.8 */
9. **if** the root has only one child /* and it is not a leaf */
10. Remove the root
11. Set as new root its only child
12. **endif**

Fig. 1.7. The R-tree deletion algorithm.

Regarding the deletion of an entry from an R-tree, it is performed with the algorithm given in Figure 1.7. We note that the handling of an underflowing node (a node with fewer than m entries) is different in the R-tree, compared

```
Algorithm CondenseTree(TypeNode L)
/* Given is the leaf L from which an entry E has been deleted. If after
the deletion of E, L has fewer than m entries, then remove entirely
leaf L and reinsert all its entries. Updates are propagated upwards and
the MBRs in the path from root to L are modified (possibly become smaller) */

1.  Set X = L
2.  Let N be the set of nodes that are going to be removed from
    the tree (initially, N is empty)
3.  while X is not the root
4.      Let Parent_X be the father node of X
5.      Let E_X be the entry of Parent_X that corresponds to X
6.      if X contains less than m entries
7.          Remove E_X from Parent_X
8.          Insert X into N
9.      endif
10.     if X has not been removed
11.         Adjust its corresponding MBR E_X.mbr, so as to enclose
            all rectangles in X /* E_X.mbr may become smaller */
12.     endif
13.     Set X = Parent_X
14. endwhile
15. Reinsert all the entries of nodes that are in the set N
```

Fig. 1.8. The R-tree condense algorithm.

with the case of B^+-tree. In the latter, an underflowing case is handled by merging two sibling nodes. Since B^+-trees index one-dimensional data, two sibling nodes will contain consecutive entries. However, for multi-dimensional data, this property does not hold. Although one still may consider promising the merging of two R-tree nodes that are stored at the same level, reinsertion is more appealing for the following reasons:

- Reinsertion achieves the same result as merging. Additionally, the algorithm for insertion is used. Also, as the number of disk accesses during the deletion operation is crucial for its performance, we have to notice that the pages required during reinsertion are available in the buffer memory, because they were retrieved during the searching of the deleted entry.
- As described, the Insert algorithm tries to maintain the good quality of the tree during the query operations. Therefore, it sounds reasonable to use reinsertion, because the quality of the tree may decrease after several deletions.

In all R-tree variants that have appeared in the literature, tree traversals for any kind of operations are executed in exactly the same way as in the original R-tree. Basically, the variations of R-trees differ in how they perform splits during insertion by considering different minimization criteria instead of the sum of the areas of the two resulting nodes.

1.2 Summary

The original R-tree structure proposed by Guttman in [81] aimed at efficient management of large collections of two-dimensional rectangles in VLSI applications. The R-tree is a dynamic access method that organizes the data objects by means of a hierarchical organization of rectangles. The structure supports insertions, deletions, and queries and uses several heuristics to minimize the overlapping of MBRs and reduce their size. These two properties are fundamental to efficient query processing, because the performance of a query is analogous to the number of node accesses required to determine the answer.

Now, R-trees are found everywhere. Several modifications to the original structure have been proposed to either improve its performance or adapt the structure in a different application domain. Based on this fact, the next two chapters are devoted to the presentation and annotation of R-tree variations. The number of the R-tree variants is quite large, so we examine them in several subsections, having in mind the special characteristics of the assumed environment or application. Chapters 4 and 5 focus on query processing issues by considering new types of queries, such as topological, directional, categorical, and distance-based. Chapters 6 and 7 present the use of R-tree variations in advanced applications such as multimedia databases, data warehousing, and data mining. Query optimization issues are covered Chapter 8. Analytical cost models and histogram-based techniques are described. Finally, Chapter 9 describes implementation issues concerning R-trees, such as parallelism and concurrency control, and summarizes what is known from the literature about prototype and commercial systems that have implemented them. The Epilogue concludes the work and gives some directions for further investigation.