2. Dynamic Versions of R-trees

The survey by Gaede and Guenther [69] annotates a vast list of citations related to multi-dimensional access methods and, in particular, refers to R-trees to a significant extent. In this chapter, we are further focusing on the family of R-trees by enlightening the similarities and differences, advantages and disadvantages of the variations in a more exhaustive manner. As the number of variants that have appeared in the literature is large, we group them according to the special characteristics of the assumed environment or application, and we examine the members of each group.

In this chapter, we present dynamic versions of the R-tree, where the objects are inserted on a one-by-one basis, as opposed to the case where a special packing technique can be applied to insert an a priori known static set of objects into the structure by optimizing the storage overhead and the retrieval performance. The latter case will be examined in the next chapter. In simple words, here we focus on the way dynamic insertions and splits are performed in assorted R-tree variants.

2.1 The R^+-tree

The original R-tree has two important disadvantages that motivated the study of more efficient variations:

1. The execution of a point location query in an R-tree may lead to the investigation of several paths from the root to the leaf level. This characteristic may lead to performance deterioration, specifically when the overlap of the MBRs is significant.
2. A few large rectangles may increase the degree of overlap significantly, leading to performance degradation during range query execution, due to empty space.

R^+-trees were proposed as a structure that avoids visiting multiple paths during point location queries, and thus the retrieval performance could be improved [211, 220]. Moreover, MBR overlapping of internal modes is avoided. This is achieved by using the clipping technique. In simple words, R^+-trees do not allow overlapping of MBRs at the same tree level. In turn, to achieve this, inserted objects have to be divided in two or more MBRs, which means that a

specific object's entries may be duplicated and redundantly stored in several nodes.

Figure 2.1 demonstrates an R$^+$-tree example. Although the structure looks similar to that of the R-tree, notice that object d is stored in two leaf nodes B and C. Also, notice that due to clipping no overlap exists between nodes at the same level.

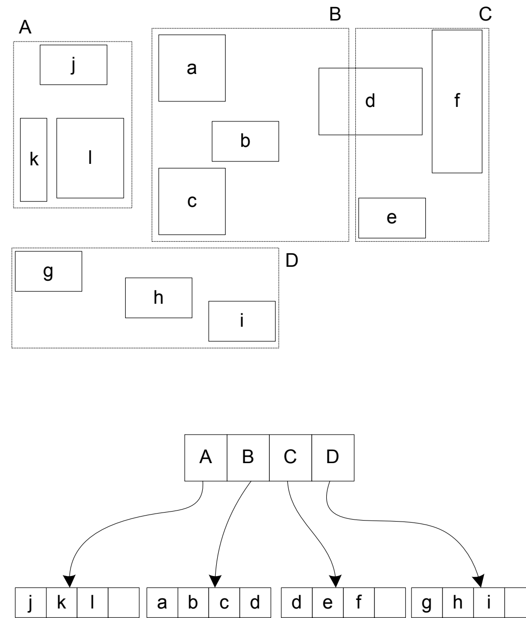


Fig. 2.1. An R$^+$-tree example.

The algorithm for range query processing is similar to the one used for R-trees. The only difference is that duplicate elimination is necessary to avoid reporting an object more than once. However, insertion, deletion, and node splitting algorithms are different due to the clipping technique applied. In order to insert a new entry E, the insertion algorithm starts from the root and determines the MBRs that intersect $E.mbr$. Then $E.mbr$ is clipped and the procedure is recursively applied for the corresponding subtrees. The insertion algorithm is given in Figure 2.2.

The fact that multiple copies of an object's MBR may be stored in several leaf nodes has a direct impact on the deletion algorithm. All copies of an object's MBR must be removed from the corresponding leaf nodes. The deletion algorithm is illustrated in Figure 2.3.

Evidently, an increased number of deletions may reduce storage utilization significantly. Therefore, appropriate reorganization must be performed to handle underutilized tree nodes. The CondenseTree algorithm, already illustrated in Figure 1.8 can be used for this purpose.

```
Algorithm Insert(TypeEntry E, TypeNode RN)
/* Inserts a new entry E in the R+-tree rooted at node RN */

1.  if RN is not a leaf node
2.      foreach entry e of RN
3.          if e.mbr overlaps with E.mbr
4.              Call Insert(E, e.ptr)
5.          endif
6.      endfor
7.  else // RN is a leaf node
8.      if there is available space in RN
9.          Add E to RN
10.     else
11.         Call SplitNode(RN)
12.         Perform appropriate tree reorganization to reflect changes
13.     endif
14. endif
```

Fig. 2.2. The R$^+$-tree insertion algorithm.

```
Algorithm Delete(TypeEntry E, TypeNode RN)
/* Deletes an existing entry E from the R+-tree rooted at node RN */

1.  if RN is not a leaf node
2.      foreach entry e of RN
3.          if e.mbr overlaps with E.mbr
4.              Call Delete (E,e.ptr)
5.              Calculate the new MBR of the node
6.              Adjust the MBR of the parent node accordingly
7.          endif
8.      endfor
9.  else // RN is a leaf node
10.     Remove E from RN.
11.     Adjust the MBR of the parent node accordingly
12. endif
```

Fig. 2.3. The R$^+$-tree deletion algorithm.

During the execution of the insertion algorithm a node may become full, and therefore no more entries can be stored in it. To handle this situation, a node splitting mechanism is required as in the R-tree case. The main difference between the R$^+$-tree splitting algorithm and that of the R-tree is that downward propagation may be necessary, in addition to the upward propagation. Recall that in the R-tree case, upward propagation is sufficient to guarantee the structure's integrity.

Therefore, this redundancy works in the opposite direction of decreasing the retrieval performance in case of window queries. At the same time, another side effect of clipping is that during insertions, an MBR augmentation may lead

to a series of update operations in a chain reaction type. Also, under certain circumstances, the structure may lead to a deadlock, as, for example, when a split has to take place at a node with $M+1$ rectangles, where every rectangle encloses a smaller one.

2.2 The R*-tree

R*-trees [19] were proposed in 1990 but are still very well received and widely accepted in the literature as a prevailing performance-wise structure that is often used as a basis for performance comparisons. As already discussed, the R-tree is based solely on the area minimization of each MBR. On the other hand, the R*-tree goes beyond this criterion and examines several others, which intuitively are expected to improve the performance during query processing. The criteria considered by the R*-tree are the following:

Minimization of the area covered by each MBR. This criterion aims at minimizing the dead space (area covered by MBRs but not by the enclosed rectangles), to reduce the number of paths pursued during query processing. This is the single criterion that is also examined by the R-tree.

Minimization of overlap between MBRs. Since the larger the overlapping, the larger is the expected number of paths followed for a query, this criterion has the same objective as the previous one.

Minimization of MBR margins (perimeters). This criterion aims at shaping more quadratic rectangles, to improve the performance of queries that have a large quadratic shape. Moreover, since quadratic objects are packed more easily, the corresponding MBRs at upper levels are expected to be smaller (i.e., area minimization is achieved indirectly).

Maximization of storage utilization. When utilization is low, more nodes tend to be invoked during query processing. This holds especially for larger queries, where a significant portion of the entries satisfies the query. Moreover, the tree height increases with decreasing node utilization.

The R*-tree follows an engineering approach to find the best possible combinations of the aforementioned criteria. This approach is necessary, because the criteria can become contradictory. For instance, to keep both the area and overlap low, the lower allowed number of entries within a node can be reduced. Therefore, storage utilization may be impacted. Also, by minimizing the margins so as to have more quadratic shapes, the node overlapping may be increased.

For the insertion of a new entry, we have to decide which branch to follow, at each level of the tree. This algorithm is called ChooseSubtree. For instance, we consider the R*-tree depicted in Figure 2.4. If we want to insert data rectangle r, ChooseSubtree commences from the root level, where it chooses the entry whose MBR needs the least area enlargement to cover r. The required node is N_5 (for which no enlargement is needed at all). Notice that the examined criterion is *area minimization*. For the leaf nodes, ChooseSubtree considers

the *overlapping minimization* criterion, because experimental results in [19] indicate that this criterion performs slightly better than others. Therefore, for the subtree rooted at N_5, ChooseSubtree selects the entry whose MBR enlargement leads to the smallest overlap increase among the sibling entries in the node, that is, node N_1.

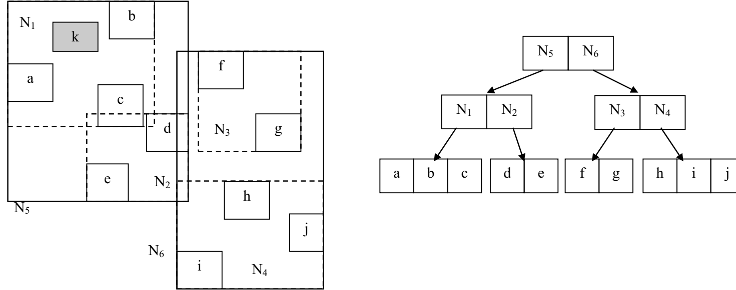


Fig. 2.4. An R*-tree example.

In case ChooseSubtree selects a leaf that cannot accommodate the new entry (i.e., it already has the maximum number of entries), the R*-tree does not immediately resort to node splitting. Instead, it finds a fraction of the entries from the overflowed node and reinserts them. The set of entries to be reinserted are those whose centroid distances from node centroid are among the largest 30% (i.e., this is a heuristic to detect and discard the furthest entries). In the example of Figure 2.4, assume that N_1 is overflowed. In this case, entry b is selected for reinsertion, as its centroid is the farthest from the centroid of N_1.

The reinsertion algorithm achieves a kind of tree rebalancing and significantly improves performance during query processing. However, reinsertion is a costly operation. Therefore, only one application of reinsertion is permitted for each level of the tree. When overflow cannot be handled by reinsertion, node splitting is performed; the Split algorithm consists of two steps. The first step decides a split axis among all dimensions. The split axis is the one with the smallest overall perimeter; it works by sorting all the entries by the coordinates of their left boundaries. Then it considers every division of the sorted list that ensures that each node is at least 40% full. Assume that we have to split the node depicted in Figure 2.5 (we assume that a node can have a single entry, which corresponds to 33% utilization). The 1-3 division (Figure 2.5a), allocates the first entry into N and the other 3 entries into N'. The algorithm computes the perimeters of N and N' and performs the same computation for the other (2-2, 3-1) divisions. A second pass repeats this process with respect to the MBRs' right boundaries. Finally, the overall perimeter on the x-axis equals the sum of all the perimeters obtained from the two passes.

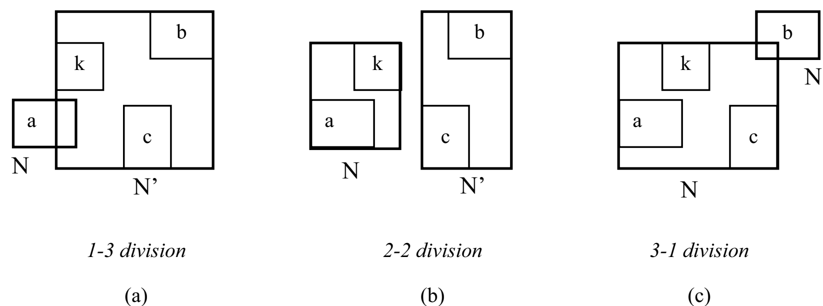


Fig. 2.5. An example of examined divisions, when split axis is x.

When the split axis is selected, the split algorithm sorts the entries (according to their lower or upper boundaries) on the selected dimension, and again, examines all possible divisions. The final division is the one that has the minimum overlap between the MBRs of the resulting nodes. Continuing the previous example, assume that the split axis is x. Then, among the possible divisions 2-2 incurs zero overlap (between N and N') and thus becomes the final splitting. The cost of the split algorithm of the R*-tree is computed as follows. Let M and m be the maximum and minimum allowed number of entries, respectively. The entries are sorted twice, with cost $O(M \log M)$. For each axis, the margin of $2 \times 2 \times (M - 2m + 2)$ rectangles and the overlap of $2 \times (M - 2m + 2)$ divisions is calculated.

Finally, we notice that the R*-tree does not use any specialized deletion algorithm. Instead, deletion in the R*-tree is performed with the deletion algorithm of the original R-tree.

2.3 The Hilbert R-tree

The Hilbert R-tree [105] is a hybrid structure based on the R-tree and the B$^+$-tree. Actually, it is a B$^+$-tree with geometrical objects being characterized by the Hilbert value of their centroid. The structure is based on the Hilbert space-filling curve . It has been shown in [97, 152] that the Hilbert space-filling curve preserves well the proximity of spatial objects. Figure 2.6 illustrates three space-filling curves.

Entries of internal tree nodes are augmented by the largest Hilbert value of their descendants. Therefore, an entry e of an internal node is a triplet (mbr, H, p) where mbr is the MBR that encloses all the objects in the corresponding subtree, H is the maximum Hilbert value of the subtree, and p is the pointer to the next level. Entries in leaf nodes are exactly the same as in R-trees, R$^+$-trees, and R*-trees and are of the form (mbr, oid), where mbr is the MBR of the object and oid the corresponding object identifier.

The algorithm for range query processing is the same as that of the R-tree and the R*-tree. Staring from the root, it descents the tree by checking if the

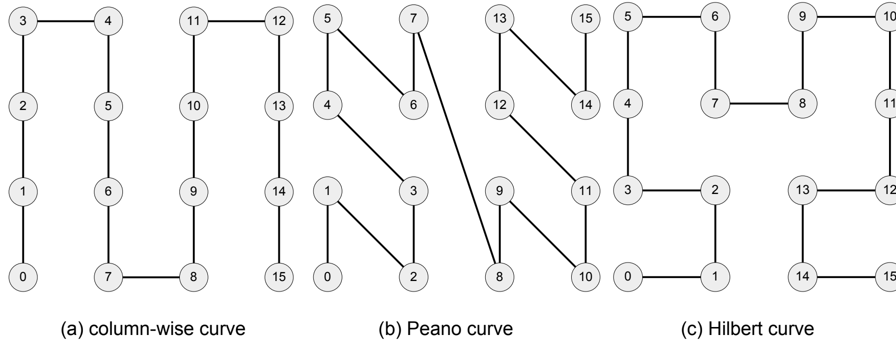


Fig. 2.6. Three space-filling curves.

query rectangle intersects with the MBRs of the accessed nodes. When a leaf is reached, all relevant objects are reported. However, insertion in a Hilbert R-tree differs significantly from that of other variants. In order to insert a new entry E, the Hilbert value H of its rectangle centroid is calculated. Then, H is used as a key that guides the insertion process. At each node, the Hilbert values of the alternative subtrees are checked and the smallest one that is larger than H is selected. The details of the insertion algorithm are given in Figure 2.7.

Algorithm Insert(TypeEntry E**, TypeNode** RN**)**
/* Inserts a new entry E in the Hilbert R-tree rooted at node RN */

1. **if** RN is not a leaf node
2. Among the entries of the node select the entry e that has the smallest H value and is larger than the Hilbert value of the new object
3. Call **Insert(**E**,** $e.ptr$**)**
4. Adjust MBR of parent
5. **else** // RN is a leaf node
6. **if** there is available space in RN
7. Add E to RN in a position respecting the Hilbert order
8. **else**
9. Call **HandleOverflow(**E,RN**)**
10. Propagate changes upwards
11. **endif**
12. **endif**

Fig. 2.7. The Hilbert R-tree insertion algorithm.

An important characteristic of the Hilbert R-tree that is missing from other variants is that there exists an order of the nodes at each tree level, respecting the Hilbert order of the MBRs. This order allows defining siblings for each tree node, as in the case of the B$^+$-tree. The existence of siblings enables the

delay of a node split when this node overflows. Instead of splitting a node immediately after its capacity has been exceeded, we try to store some entries in sibling nodes. A split takes place only if all siblings are also full. This unique property of the Hilbert R-tree helps considerably in storage utilization increase, and avoids unnecessary split operations. The decision to perform a split is controlled by the HandleOverflow algorithm, which is illustrated in Figure 2.8. In the case of a split, a new node is returned by the algorithm.

Algorithm HandleOverflow(TypeEntry E, TypeNode RN)
/* Takes care of the overflowing node RN upon insertion of E.
Returns a new node NN in the case of a split, and NULL otherwise */

1. Let $\mathcal{E}$ denote the set of entries of node RN and its s-1 sibling nodes
2. Set $\mathcal{E} = \mathcal{E} \cup E$
3. **if** there exists a node among the s-1 siblings that is not full
4. Distribute all entries in $\mathcal{E}$ among the s nodes respecting the Hilbert ordering
5. Return **NULL**
6. **else** // all s-1 siblings are full
7. Create a new node NN
8. Distribute all entries in $\mathcal{E}$ among the s+1 nodes
9. Return the new node NN
10. **endif**

Fig. 2.8. The Hilbert R-tree overflow handling algorithm.

A split takes place only if all s siblings are full, and thus s+1 nodes are produced. This heuristic is similar to that applied in B*-trees, where redistribution and 2-to-3 splits are performed during node overflows [111].

It is evident that the Hilbert R-tree acts like a B$^+$-tree for insertions and like an R-tree for queries. According to the authors' experimentation in [105], Hilbert R-trees were proven to be the best dynamic version of R-trees as of the time of publication. However, this variant is vulnerable performance-wise to large objects. Moreover, by increasing the space dimensionality, proximity is not preserved adequately by the Hilbert curve, leading to increased overlap of MBRs in internal tree nodes.

2.4 Linear Node Splitting

Ang and Tan in [11] have proposed a linear algorithm to distribute the objects of an overflowing node in two sets. The primary criterion of this algorithm is to distribute the objects between the two nodes as evenly as possible, whereas the second criterion is the minimization of the overlapping between them. Finally, the third criterion is the minimization of the total coverage.

In order to minimize overlapping, all rectangles within an overflowed node are tried to be pushed as far apart as possible to the boundary of the overflowed node's MBR. Let each rectangle be denoted as (xl, yl, xh, yh), whereas the MBR of an overflowed node N is denoted $R_N = (L, B, R, T)$. The algorithm in [11] uses four lists, denoted $LIST_L$, $LIST_B$, $LIST_R$, $LIST_T$, which store the rectangles of N that are nearer to the corresponding border than to its opposite (borders are considered in pairs: left-right, bottom-top). The algorithm is given in Figure 2.9.

Algorithm NewLinear(TypeNode N)

1. Set $\text{LIST}_L \leftarrow \text{LIST}_R \leftarrow \text{LIST}_B \leftarrow \text{LIST}_T \leftarrow \emptyset$
2. **foreach** rectangle $S = (xl, yl, xh, yh)$ in N with $R_N = (L, B, R, T)$
3.   **if** $xl - L < R - xh$
4. $\text{LIST}_L \leftarrow \text{LIST}_L \cup S$
5.   **else**
6. $\text{LIST}_R \leftarrow \text{LIST}_R \cup S$
7.   **endif**
8.   **if** $xy - B < T - yh$
9. $\text{LIST}_B \leftarrow \text{LIST}_B \cup S$
10.   **else**
11. $\text{LIST}_T \leftarrow \text{LIST}_T \cup S$
12.   **endif**
13. **endfor**
14. **if** $\max(|\text{LIST}_L|, |\text{LIST}_R|) < \max(|\text{LIST}_B|, |\text{LIST}_T|)$
15.   Split the node along the x direction
16. **else if** $\max(|\text{LIST}_L|, |\text{LIST}_R|) > \max(|\text{LIST}_B|, |\text{LIST}_T|)$
17.   Split the node along the y direction
18. **else** //tie break
19.   **if** overlap(LIST_L, LIST_R) $<$ overlap(LIST_B, LIST_T)
20.   Split the node along the x direction
21.   **else if** overlap(LIST_L, LIST_R) $>$ overlap(LIST_B, LIST_T)
22.   Split the node along the y direction
23.   **else**
24.   Split the node along the direction with smallest total coverage
25.   **endif**
26. **endif**

Fig. 2.9. The Linear Node Split algorithm.

It is easy to notice that each rectangle in N will participate either in $LIST_L$ or $LIST_R$. The same applies for $LIST_B$ or $LIST_T$. The decision to carry the split along the horizontal or vertical axis depends on the distribution of the rectangles (the corresponding lines in the code are the ones that compare the maxima among sizes of the lists). In case of a tie, the algorithm resorts to the minimum overlapping criterion.

As mentioned in [11], the algorithm may have a disadvantage in the case that most rectangles in node N (the one to be split) form a cluster, whereas a few others are outliers. This is because then the sizes of the lists will be highly skewed. As a solution, Ang and Tan proposed reinsertion of outliers.

Experiments using this algorithm have shown that it results in R-trees with better characteristics and better performance for window queries in comparison with the quadratic algorithm of the original R-tree.

2.5 Optimal Node Splitting

As has been described in Section 2.1, three node splitting algorithms were proposed by Guttman to handle a node overflow. The three algorithms have linear, quadratic, and exponential complexity, respectively. Among them, the exponential algorithm achieves the optimal bipartitioning of the rectangles, at the expense of increased splitting cost. On the other hand, the linear algorithm is more time efficient but fails to determine an optimal rectangle bipartition. Therefore, the best compromise between efficiency and bipartition optimality is the quadratic algorithm.

Garcia, Lopez, and Leutenegger elaborated the optimal exponential algorithm of Guttman and reached a new optimal polynomial algorithm $O(n^d)$, where d is the space dimensionality and $n = M + 1$ is the number of entries of the node that overflows [71]. For n rectangles the number of possible bipartitions is exponential in n. Each bipartition is characterized by a pair of MBRs, one for each set of rectangles in each partition. The key issue, however, is that a large number of candidate bipartitions share the same pair of MBRs. This happens when we exchange rectangles that do not participate in the formulation of the MBRs. The authors show that if the cost function used depends only on the characteristics of the MBRs, then the number of different MBR pairs is polynomial. Therefore, the number of different bipartitions that must be evaluated to minimize the cost function can be determined in polynomial time.

The proposed optimal node splitting algorithm investigates each of the $O(n^2)$ pairs of MBRs and selects the one that minimizes the cost function. Then each one of the rectangles is assigned to the MBR that it is enclosed by. Rectangles that lie at the intersection of the two MBRs are assigned to one of them according to a selected criterion.

In the same paper, the authors give another insertion heuristic, which is called *sibling-shift*. In particular, the objects of an overflowing node are optimally separated in two sets. Then one set is stored in the specific node, whereas the other set is inserted in a sibling node that will depict the minimum increase of an objective function (e.g., expected number of disk accesses). If the latter node can accommodate the specific set, then the algorithm terminates. Otherwise, in a recursive manner the latter node is split. Finally, the process terminates when either a sibling absorbs the insertion or this is not possible, in which case a new node is created to store the pending set. The

authors reported that the combined use of the optimal partitioning algorithm and the sibling-shift policy improved the index quality (i.e., node utilization) and the retrieval performance in comparison to the Hilbert R-trees, at the cost of increased insertion time. This has been demonstrated by an extensive experimental evaluation with real-life datasets.

2.6 Branch Grafting

More recently, in [208] an insertion heuristic was proposed to improve the shape of the R-tree so that the tree achieves a more elegant shape, with a smaller number of nodes and better storage utilization. In particular, this technique considers how to redistribute data among neighboring nodes, so as to reduce the total number of created nodes. The approach of branch grafting is motivated by the following observation. If, in the case of node overflow, we examined all other nodes to see if there is another node (at the same level) able to accommodate one of the overflowed node's rectangles, the split could be prevented. Evidently, in this case, a split is performed only when all nodes are completely full. Since the aforementioned procedure is clearly prohibitive as it would dramatically increase the insertion cost, the branch grafting method focuses only on the neighboring nodes to redistribute an entry from the overflowed node. Actually, the term *grafting* refers to the operation of moving a leaf or internal node (along with the corresponding subtree) from one part of the tree to another.

The objectives of branch grafting are to achieve better-shaped R-trees and to reduce the total number of nodes. Both these factors can improve performance during query processing. To illustrate these issues, the following example is given in [208]. Assume that we are inserting eight rectangles (with the order given by their numbering), which are depicted in Figure 2.10. Let the maximum (minimum) number of entries within a node be equal to 4 (2). Therefore, the required result is shown in Figure 2.10(a), because they clearly form two separate groups. However, by using theR-tree insertion algorithm, which invokes splitting after each overflow, the result in Figure 2.10(b) would be produced.

Using the branch and grafting method, the split after the insertion of rectangle H can be avoided. Figure 2.11(a) illustrates the resulted R-tree after the insertion of the first seven rectangles (i.e., A to G). When rectangle H has to be inserted, the branch grafting method finds out that rectangle 3 is covered by node R_1, which has room for one extra rectangle. Therefore, rectangle C is moved from node R_2 to node R_1, and rectangle H can be inserted in R_2 without causing an overflow. The resulted R-tree is depicted in Figure 2.11(b).

In summary, in case of node overflow, the branch grafting algorithm first examines the parent node, to find the MBRs that overlap the MBR of the overflowed node. Next, individual records in the overflowed are examined to see if they can be moved to the nodes corresponding to the previously found overlapping MBRs. Records are moved only if the resulting area of coverage for

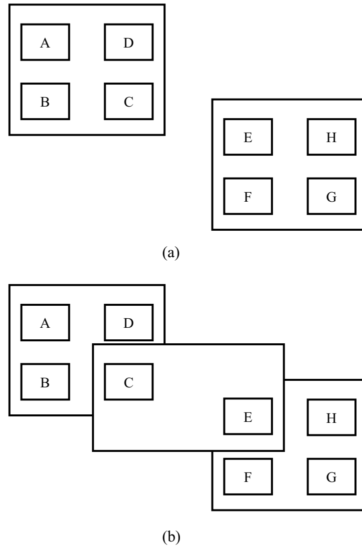


Fig. 2.10. Branch grafting technique: (a) Optimal result after inserting 8 rectangles; (b) actual result produced after two R-tree splits.

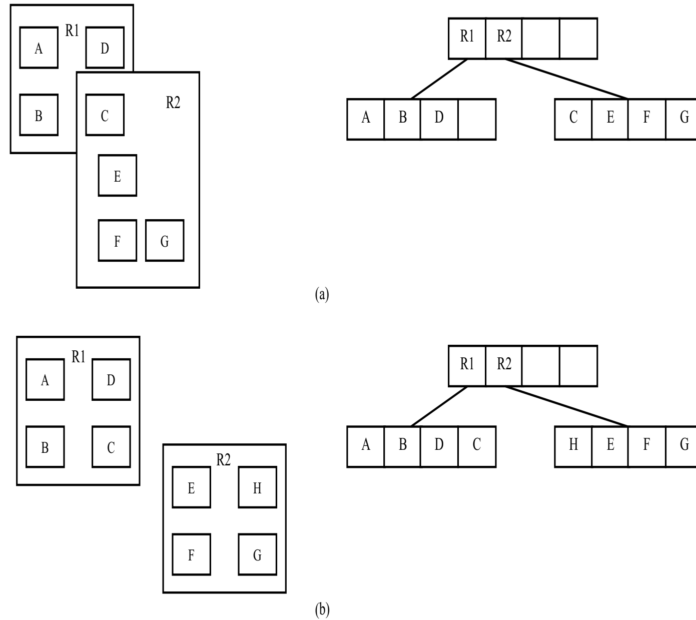


Fig. 2.11. (a) Resulted R-tree after the first 7 insertions (rectangles A to G); (b) result of branch grafting after inserting the rectangle H.

the involved nodes does not have to be increased after the moving of records. In the case that no movement is possible, a normal node split takes place.

In general, the approach of branch grafting has some similarities with the forced reinsertion, which is followed by the R*-tree. Nevertheless, as mentioned in [208], branch grafting is not expected to outperform forced reinsertion during query performance. However, one may expect that, because branch grafting tries to locally handle the overflow, the overhead to the insertion time will be smaller than that of forced reinsertion. In [208], however, the comparison considers only various storage utilization parameters, not query processing performance.

2.7 Compact R-trees

Huang et al. proposed Compact R-trees, a dynamic R-tree version with optimal space overhead [93]. The motivation behind the proposed approach is that R-trees, R$^+$-trees, and R*-trees suffer from the storage utilization problem, which is around 70% in the average case. Therefore, the authors improve the insertion mechanism of R-trees to a more compact R-tree structure, with no penalty on performance during queries.

The heuristics applied are simple, meaning that no complex operations are required to significantly improve storage utilization. Among the $M+1$ entries of an overflowing node during insertions, a set of M entries is selected to remain in this node, under the constraint that the resulting MBR is the minimum possible. Then the remaining entry is inserted to a sibling that:

- has available space, and
- whose MBR is enlarged as little as possible.

Thus, a split takes place only if there is no available space in any of the sibling nodes.

Performance evaluation results reported in [93] have shown that the storage utilization of the new heuristic is between 97% and 99%, which is a great improvement. A direct impact of the storage utilization improvement is the fact that fewer tree nodes are required to index a given dataset. Moreover, less time is required to build the tree by individual insertions, because of the reduced number of split operations required. Finally, caching is improved because the buffer associated with the tree requires less space to accommodate tree nodes. It has been observed that the query performance of window queries is similar to that of Guttman's R-tree.

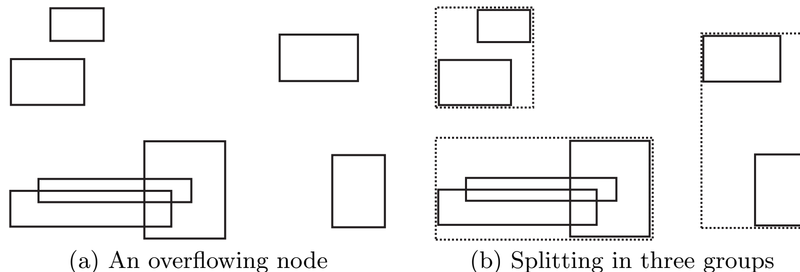
2.8 cR-trees

Motivated by the analogy between separating of R-tree node entries during the split procedure on the one hand and clustering of spatial objects on the

other hand, Brakatsoulas et al. [32] have altered the assumption that an R-tree overflowing node has to necessarily be split in exactly two nodes. In particular, using the k-means clustering algorithm as a working example, they implemented a novel splitting procedure that results in up to k nodes ($k \geq 2$ being a parameter).

In fact, R-trees and their variations (R*-trees, etc.) have used heuristic techniques to provide an efficient splitting of $M+1$ entries of a node that overflows into two groups (minimization of area enlargement, minimization of overlap enlargement, combinations, etc.). On the other hand, Brakatsoulas et al. observed that node splitting is an optimization problem that takes a local decision according to the objective that the probability of simultaneous access to the resulting nodes after split is minimized during a query operation. Indeed, clustering maximizes the similarity of spatial objects within each cluster (intracluster similarity) and minimizes the similarity of spatial objects across clusters (intercluster similarity). The probability of accessing two node rectangles during a selection operation (hence, the probability of traversing two subtrees) is proportional to their similarity. Therefore, node splitting should (a) assign objects with high probability of simultaneous access to the same node and (b) assign objects with low probability of simultaneous access to different nodes. Taking this into account, the authors considered the R-tree node splitting procedure as a typical clustering problem of finding the "optimal" k clusters of $n = M + 1$ entries ($k \geq 2$).

The well-known k-means clustering algorithm was chosen to demonstrate the efficiency of the cR-tree. According to the authors of the paper, the choice was based on the efficiency of the k-means algorithm ($O(kn)$ time complexity and $O(n + k)$ space complexity, analogous to the R-tree Linear split algorithm) and its order independence (unlike Guttman's linear split algorithm). However, a disadvantage of the chosen algorithm is the fact that it requires k to be given as input. Apparently, the "optimal" number of nodes after splitting is not known in advance. To overcome it, the authors adopted an incremental approach: starting from k=2 and increasing k by one each time, they compared the clustering quality of two different clusterings Cluster$(M + 1, k)$ and Cluster$(M + 1, k + 1)$ using average silhouette width, a clustering quality measure originally proposed by Kaufman and Rousseeuw [109]. In practice, the



(a) An overflowing node (b) Splitting in three groups

Fig. 2.12. Splitting an R-tree node in more than two nodes.

theoretical limit of $k_{max} = M + 1$ was bounded to $k_{max} = 5$ as a safe choice. An example is illustrated in Figure 2.12, where the entries of an overflowing node are distributed in three groups.

The empirical studies provided in the paper illustrate that the cR-tree query performance was competitive with the R*-tree and was much better than that of the R-tree. Considering index construction time, the cR-tree was shown to be at the efficiency level of the R-tree and much faster than the R*-tree.

2.9 Deviating Variations

Apart from the aforementioned R-tree variations, a number of interesting extensions and adaptations have been proposed that in some sense deviate drastically from the original idea of R-trees. Among other efforts, we note the following research works.

The Sphere-tree by Oosterom uses minimum bounding spheres instead of MBRs [164], whereas the Cell-tree by Guenther uses minimum bounding polygons designed to accommodate arbitrary shape objects [77]. The Cell-tree is a clipping-based structure and, thus, a variant has been proposed to overcome the disadvantages of clipping. The latter variant uses "oversize shelves", i.e., special nodes attached to internal ones, which contain objects that normally should cause considerable splits [78, 79].

Similarly to Cell-trees, Jagadish and Schiwietz proposed independently the structure of Polyhedral trees or P-trees, which use minimum bounding polygons instead of MBRs [96, 206].

The QR-tree proposed in [146] is a hybrid access method composed of a Quadtree [203] and a forest of R-trees. The Quadtree is used to decompose the data space into quadrants. An R-tree is associated with each quadrant, and it is used to index the objects' MBRs that intersect the corresponding quadrant. Performance evaluation results show that for several cases the method performs better than the R-tree. However, large objects intersect many quadrants and therefore the storage requirements may become high due to replication.

The S-tree by Aggarwal et al. relaxes the rule that the R-tree is a balanced structure and may have leaves at different tree levels [5]. However, S-trees are static structures in the sense that they demand the data to be known in advance.

A number of recent research efforts shares a common idea: to couple the R-tree with an auxiliary structure, thus sacrificing space for the performance. Ang and Tan proposed the Bitmap R-tree [12], where each node contains bitmap descriptions of the internal and external object regions except the MBRs of the objects. Thus, the extra space demand is paid off by savings in retrieval performance due to better tree pruning. The same trade-off holds for the RS-tree, which is proposed by Park et al. [184] and connects an R*-tree with a signature tree with a one-to-one node correspondence. Lastly, Lee

and Chung [133] developed the DR-tree, which is a main memory structure for multi-dimensional objects. They couple the R*-tree with this structure to improve the spatial query performance.

Bozanis et al. have partitioned the R-tree in a number of smaller R-trees [31], along the lines of the binomial queues that are an efficient variation of heaps.

Agarwal et al. [4] proposed the Box-tree, that is, a bounding-volume hierarchy that uses axis-aligned boxes as bounding volumes. They provide worst-case lower bounds on query complexity, showing that Box-trees are close to optimal, and they present algorithms to convert Box-trees to R-trees, resulting in R-trees with (almost) optimal query complexity.

Recently, the optimization of data structures for cache effectiveness has attracted significant attention, and methods to exploit processors' caches have been proposed to reduce the latency that incurs by slow main memory speeds. Regular R-trees have large node size, which leads to poor cache performance. Kim et al. [110] proposed the CR-tree to optimize R-trees in main memory environments to reduce cache misses. Nodes of main memory structures are small and the tree heights are usually large. In the CR-tree, MBR keys are compressed to reduce the height of the tree. However, when transferred to a disk environment, CR-trees do not perform well. TR-trees (Two-way optimized R-trees) have been proposed by Park and Lee [185] to optimize R-trees for both cases, i.e., TR-trees are cache- and disk-optimized. In a TR-tree, each disk page maintains an R-tree-like structure. Pointers to in-page nodes are stored with fewer cost. Optimization is achieved by analytical evaluation of the cache latency cost of searching through the TR-tree, taking into account the prefetching ability of today's CPUs.

2.9.1 PR-trees

The Priority R-tree (PR-Rtree for short) has been proposed in [15] and is a provably asymptotically optimal variation of the R-tree. The term *priority* in the name of PR-tree stems from the fact that its bulk-loading algorithm utilizes the "priority rectangles".

Before describing PR-trees, Arge et al. [15] introduce pseudo-PR-trees. In a pseudo-PR-tree, each internal node v contains the MBRs of its children nodes v_c. In contrast to regular R-tree, the leaves of the pseudo-PR-tree may be stored at different levels. Also, internal nodes have degree equal to six (i.e., the maximum number of entries is six). More formally, let $\mathcal{S}$ be a set of N rectangles, i.e., $\mathcal{S} = r_1, \ldots, r_N$. Each r_i is of the form $(x_{\min}(r_i), y_{\min}(r_i)), (x_{\max}(r_i), y_{\max}(r_i))$. Similar to Hilbert R-tree, each rectangle is mapped to a 4D point, i.e., a mapping of $r_i^* = (x_{\min}(R_i), y_{\min}(R_i), x_{\max}(R_i), y_{\max}(R_i))$. Let $\mathcal{S}^*$ be the set of N resulting 4D points. A pseudo-PR-tree T_S on $\mathcal{S}$ is defined recursively [15]: if $\mathcal{S}$ contains fewer than c rectangles (c is the maximum number of rectangles that can fit in a disk page), then T_S consists of a single leaf. Otherwise, T_S consists of a node v

with six children, the four so-called priority leaves and two recursive pseudo-PR-trees. For the two recursive pseudo-PR-trees, v stores the corresponding MBRs. Regarding the construction of the priority leaves: the first such leaf is denoted as $v_p^{x_{\min}}$ and contains the c rectangles in $\mathcal{S}$ with minimal $x_{\min}$-coordinates. Analogously, the three other leaves are defined, i.e., $v_p^{y_{\min}}, v_p^{x_{\max}}$, and $v_p^{y_{\max}}$. Therefore, the priority leaves store the information on the maximum/minimum of the values of coordinates. Let $\mathcal{S}_r$ be equal to the set that results from $\mathcal{S}$ by removing the aforementioned rectangles that correspond to priority leaves. $\mathcal{S}_r$ is divided into two sets $\mathcal{S}_<$ and $\mathcal{S}_>$ of approximately the same size. The same definition is applied recursively for the two pseudo-PR-trees $\mathcal{S}_<$ and $\mathcal{S}_>$. The division is done by using in a round-robin fashion the $x_{\min}, y_{\min}, x_{\max}$, or $y_{\max}$ coordinate (a procedure that is analogous to that of building a 4D k-d-tree on $\mathcal{S}_r^*$).

The following is proven in [15]: a range query on a pseudo-PR-tree on N rectangles has I/O cost $O(\sqrt{N/c} + A/c)$, in the worst case, where A is the number of rectangles that satisfy the range query. Also, a pseudo-PR-tree can be bulk-loaded with N rectangles in the plane with worst-case I/O cost equal to $O(\frac{N}{c} \log_{M/c} \frac{N}{c})$.

A PR-tree is a height-balanced tree, i.e., all its leaves are at the same level and in each node c entries are stored. To derive a PR-tree from a pseudo-PR-tree, the PR-tree has to be built into stages, in a bottom-up fashion. First, the leaves V_0 are created and the construction proceeds to the root node. At stage i, first the pseudo-PR-tree $T_{\mathcal{S}_i}$ is constructed from the rectangles $\mathcal{S}_i$ of this level. The nodes of level i in the PR-tree consist of all the leaves of $T_{\mathcal{S}_i}$, i.e., the internal nodes are discarded. For a PR-tree indexing N rectangles, it is shown in [15] that a range query has I/O cost $O(\sqrt{N/c} + A/c)$, in the worst case, whereas the PR-tree can be bulk-loaded with N rectangles in the plane with worst-case I/O cost equal to $O(\frac{N}{c} \log_{M/c} \frac{N}{c})$. For the case of d-dimensional rectangles, it is shown that the worst-case I/O cost for the range query is $O((N/c)^{1-1/d} + A/c)$. Therefore, a PR-tree provides a worst-case optimality, because any other R-tree variant may require (in the worst case) the retrieval of leaves, even for queries that are not satisfied by any rectangle.

In addition to its worst-case analysis, the PR-tree has been examined experimentally. In summary, the bulk-loading of a PR-tree is slower than the bulk-loading of a packed 4D Hilbert tree. However, regarding query performance, for nicely distributed real data, PR-trees perform similar to existing R-tree variants. In contrast, with extreme data (very skewed data, which contain rectangles with high differences in aspect ratios), PR-trees outperform all other variants, due to their guaranteed worst-case performance.

2.9.2 LR-trees

The LR-tree [31] is an index structure based on the *logarithmic dynamization* method. LR-trees are based on decomposability. A searching problem is called *decomposable* if one can partition the input set into a set of disjoint subsets, perform the query on each subset independently, and then easily compose

the partial answers. The most useful geometric searching problems, like range searching, nearest-neighbor searching, or spatial join, as we will see, are in fact decomposable.

LR-trees consist of a number of component substructures, called *blocks*. Each block is organized as a *weak R-tree*, termed wR-tree, i.e., a semidynamic, deletions-only R-tree version. Whenever an insertion operation must be served, a set of blocks are chosen for reconstruction. On the other hand, deletions are handled locally by the block structure that accommodates the involved object. Due to the algorithms for block construction, LR-trees achieve good tightness that provides improved performance during search queries. At the same time, LR-trees maintain the efficiency of dynamic indexes during update operations.

A wR-tree on a set S of items is defined as an R-tree, which either is a legitimate created instance on only the members of S (i.e., it does not violate any of the properties of the R-tree) or is constructed from a legitimate instance on members of a set $S' \supset S$ by deleting each $x \in S' - S$ from the leaves so that: firstly, all MBRs of the structure are correctly calculated, and, secondly, the tree nodes are allowed to be underutilized (i.e., they may store fewer than m entries). The approach of wR-trees is completely opposite to the time-consuming treatment adopted by R*-trees, according to which, to strictly maintain the utilization, all underflowing nodes on the search path are deleted, and the resultant "orphaned" entries are compulsorily reinserted to the tree level that they belong.[1]

An LR-tree on a set S of N items is defined as a collection $\mathcal{C} = \{T_0, T_1, \ldots\}$ of wR-trees on a partition $V = \{V_0, V_1, \ldots\}$ of S into disjoint subsets (blocks), such that:

1. there is a "one-to-one" correspondence $V_j \leftrightarrow T_j$ between the subsets (blocks) and the elements accommodated in the trees;
2. $\left(B^{j-1} < |V_j| \leq B^{j+1}\right) \vee (|V_j| = 0)$, for some constant $B \geq 2$, which is called "base"; and
3. $|\mathcal{C}| = O(\log_B n)$.

An insertion into an LR-tree can be served by finding the first wR-tree T_j that can accommodate its own items, the items of all wR-trees to its left and the new item, then destroying all wR-trees T_k, where $k \leq j$, and finally bulk-loading a new index T_j, which stores the items in the discarded structures and the new element. This procedure is exemplified in Figure 2.13, for base $B = 2$ and node capacity $c = 4$: the LR-tree of Figure 2.13(a) accommodates 11 items. Since the binary representation of 11 is 1011, items are partitioned into three blocks, namely V_0, V_1 and V_3. Each block V_i is stored into wR-tree T_i. When the item L is inserted (Figure 2.13(b)), the cardinality of the collection becomes 12, which equals 1100 in the binary enumeration system. So the first blocks V_0, V_1 must be destroyed and replaced by a single block V_3, consisting

[1] It is important to note that B$^+$-trees follow a similar behavior in industrial applications, i.e., they do not perform node merging but only free nodes when they are completely empty.

of the elements of the set $V_0 \cup V_1 \cup \{L\}$. That change is reflected to the LR-tree, by replacing the wR-trees T_0 and T_1 by a single one T_3.

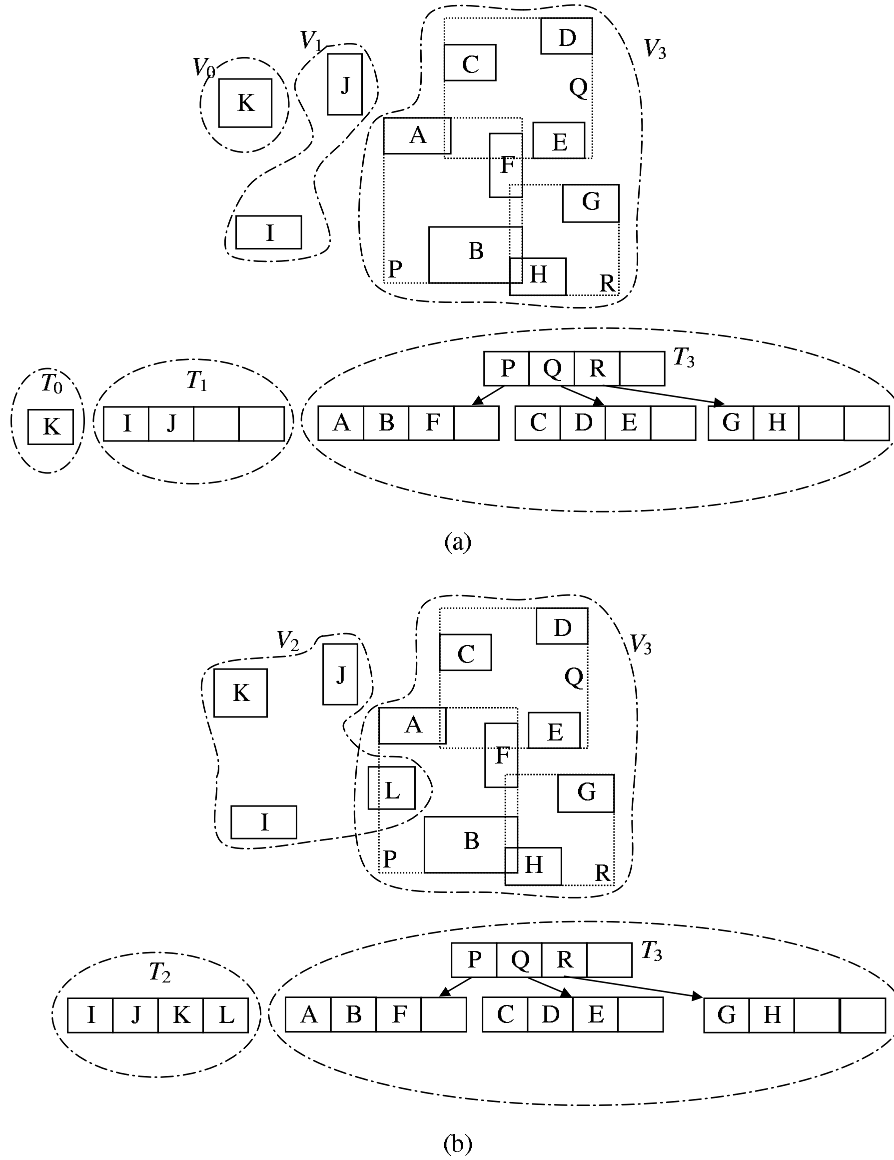


Fig. 2.13. Insertion of item L in an LR-tree with B=2. (a) The number of elements before insertion equals $11 = 1011_2$, so there are the blocks V_0, V_1, V_3 implemented as wR-trees T_0, T_1, T_3, respectively, (b) after insertion we have $12 = 1100_2$ items and therefore destruction of trees T_0, T_1 and replacement by a single tree T_2.

Algorithm RangeSearch(TypeRect Q, TypeArray $Root$)
/* Root contains the root nodes of subtrees of LR-tree */

1. $QS = \{\text{Root}[i] \mid \text{Root}[i] \cap Q \neq \emptyset\}$
2. $Answer = \bigcup_{T \in QS}$ RSearch(T,Q)
3. **return** $Answer$

Fig. 2.14. The LR-tree range search algorithm.

A range query with query rectangle Q seeks for all items whose MBRs share with Q common point(s). In LR-tree, this operation is treated by querying the individual wR-trees and concatenating the partial results trivially in O(1) time. The range search algorithm for a wR-tree T is given in Figure 2.14.

2.10 Summary

Evidently, the original R-tree, proposed by Guttman, has influenced all the forthcoming variations of dynamic R-tree structures. The R*-tree followed an engineering approach and evaluated several factors that affect the performance of the R-tree. For this reason, it is considered the most robust variant and has found numerous applications, in both research and commercial systems. However, the empirical study in [105] has shown that the Hilbert R-tree can perform better than the other variants in some cases. It is worth mentioning that the PR-tree, although a variant that deviates from other existing ones, is the first approach that offers guaranteed worst-case performance and overcomes the degenerated cases when almost the entire tree has to be traversed. Therefore, despite its more complex building algorithm, it has to be considered the best variant reported so far.