Lehrstuhl f. Dialogorientierte Systeme
Prof. Alfons Kemper, Ph.D.

Proseminar: Algorithms and Data Structures for Persistent Data

The R-Tree

Lecture and presentation: Lars Neumann
Moderation: Saskia Nieckau

June 21st , 2001
Lars Neumann
email: lars@neumann-web.de

URL of this document: http://www.fmi.uni-passau.de/~neumann/proseminar/proseminar.pdf

Content

1 Introduction

With the development of new applications in the multimedia or computer aided design (CAD) sector it became harder to handle data types such as images, audio and video files or map objects like countries, which can be found in geological databases.
An example for a geological application is, to find all farmers who own land within a particular area, e.g. if a government wants to build a new highway through this area.

Since former indexing methods (e.g. the B-Tree) are not well suited to handle multidimensional data objects of non-zero size, a new efficient and dynamic, index based structure was needed to handle this spatial data.

In 1984, Antonin Guttman presented a structure as described above; it is called the **R-Tree** (region tree). Guttman gave the algorithms for searching, updating, deleting, etc and proved the R-Tree to be very efficient [Gut84].

2 The R-Tree Index Structure

An R-Tree is similar to a B-Tree with index records in its leaf nodes (= pointers on the data objects) containing pointers to their data objects. The structure is designed so that in the case of a spatial search only a small amount of nodes has to be visited, thus it is efficient.
Because we have a completely dynamic index, insertions and deletions can be intermixed with searching. Since it is dynamic a non-periodic reorganisation becomes unnecessary.

2.1 Entries

An n-dimensional spatial database consists of an amount of tuples where each of them has a unique identifier, by which the tuple can be retrieved.
Every *leaf node* contains a tuple like

$$(I, \textit{leaf-identifier}).$$

In this case *leaf-identifier* points to the place where the object is stored in the spatial database, and I is an n-dimensional rectangle

$$I = (I_0, I_1, \ldots, I_{n-1}).$$

Every I_k represents a closed bounded interval $[a_k, b_k]$ (see figure 1), which means the

starting and ending point of the rectangle in the k-th dimension. If one or two endpoints of the interval I_k are equal to the infinity, it is meant that the described object extends outward indefinitely in the k-th dimension.
Therefore every spatial object can be represented by its *bounding box* I (also called the *minimum bounding rectangle, MBR*) (see figure 1), the smallest rectangle which contains the designated object.

Non-leaf nodes contain entries

(I, child-pointer)

where *child-pointer* points to the child node in the R-Tree and I covers all rectangles of the child node.

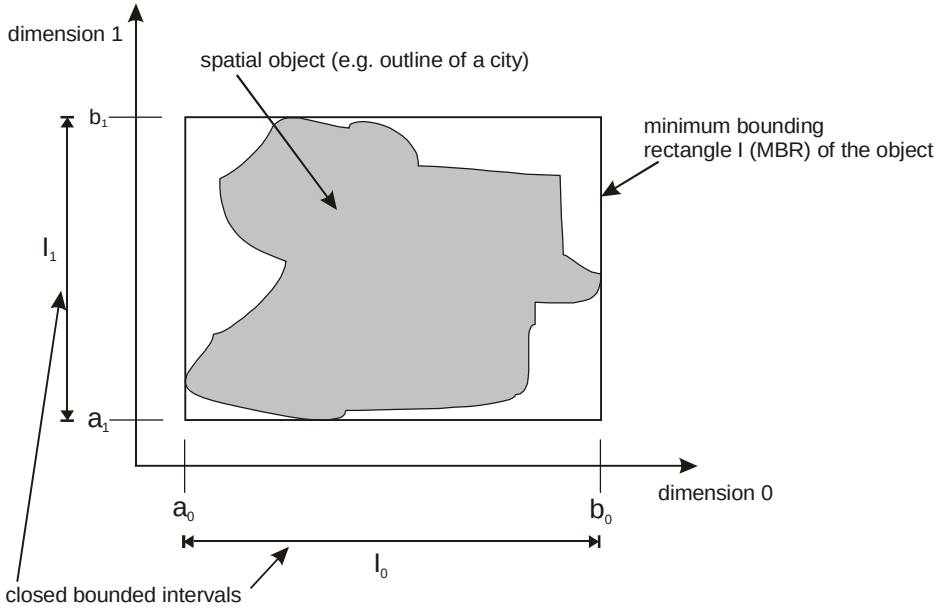

Figure 1: closed bounded interval and minimum bounding rectangle

2.2 Properties

If M (depends on the machine's disk page size and the number of dimensions n) is the maximum number of entries, that will fit in one node and $m \leq \frac{M}{2}$ specifies the minimum number of entries in a node, then a R-Tree must fulfil the following properties:

- Every leaf node, which is not the root, contains between m and M index records.
- For each index record *(I, leaf-identifier)* in a leaf, I is the smallest rectangle that contains the n-dimensional data object represented by this tuple.
- Every non-leaf node, which is not the root, has between m and M children.
- For each entry *(I, child-pointer)* in a non-leaf node, I is the smallest rectangle that contains the child node.
- The root node has at least two children unless it is a leaf.
- All leaves appear on the same level.

The height of an R-Tree, with N index records is $\lceil \log_m N \rceil - 1$ in the best case because each node contains at least m child nodes .

2.3 Overflow and Underflow

What is meant by a *node overflow*? An overflow could happen if the *m* (minimum number of entries in a node) is set too high (near to *M*), so that the node is filled very densely. If one or more additional entries will be written into this node, the maximum number of entries *M* will be exceeded and the node overflows (see figure 2).
Equivalent to the overflow the *node underflow* appears also when *m* is set too large and one or more entries are removed so that the number of entries will remain under *m* (see figure 3).

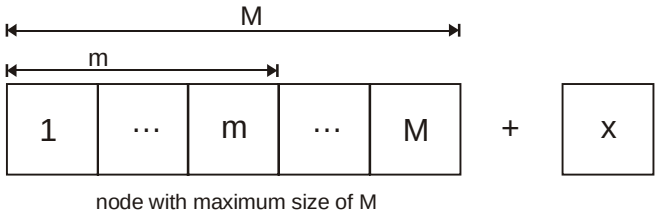

Figure 2: Node overflow

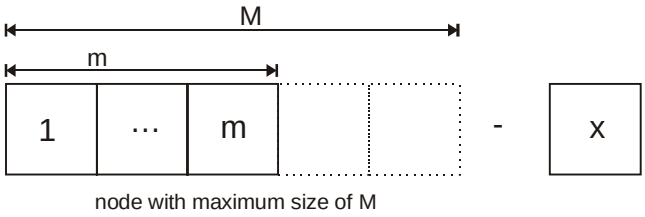

Figure 3: Node underflow

2.4 How to set *m* and *M*

The setting of *m* and *M* is very important for the efficiency of the database. While *M* depends on parameters which are given by means of the hard disk drive (HDD) properties on which the data will be stored, e.g. the disk page size or the capacity.
On the other side the size of *m* is elementary for database performance. If the database is only (mostly) needed for search inquiries with less updates, a high *m* will be recommended to keep the height of the R-Tree small and so the search performance high. But that is why the risk of an underflow or overflow is very high.
In the other case that *m* is set to a small value, the database will be good for frequent updates and modifications because the nodes are not filled so densely as if the *m* is large and the risk of over- and underflow is that obvious.

3 Algorithms

Guttman gave the algorithms for the elementary operations on the R-Tree.
These methods of the R-Tree are leant on the ones from the B-Tree, only the handling of overflow and underflow are different owing to the spatial location of the data.

In the following we define a 2-dimensional example database to describe and explain the different algorithms. The values *m* and *M* are again the minimum and the maximum amount of entries in a node. Here we set them to m=2 and M=5.
In this database (see figure 4, 5, 6) *name* is the name of the student, *semester* tells us in which semester the student is, and *credits* is the sum of all achieved credits in the university.

name	semester	credits
A	8	100
B	4	10
C	6	35
D	1	10
E	6	40
F	5	45
G	7	85
H	3	20
I	10	70
J	2	30
K	8	50
L	4	50

Figure 4: The example database

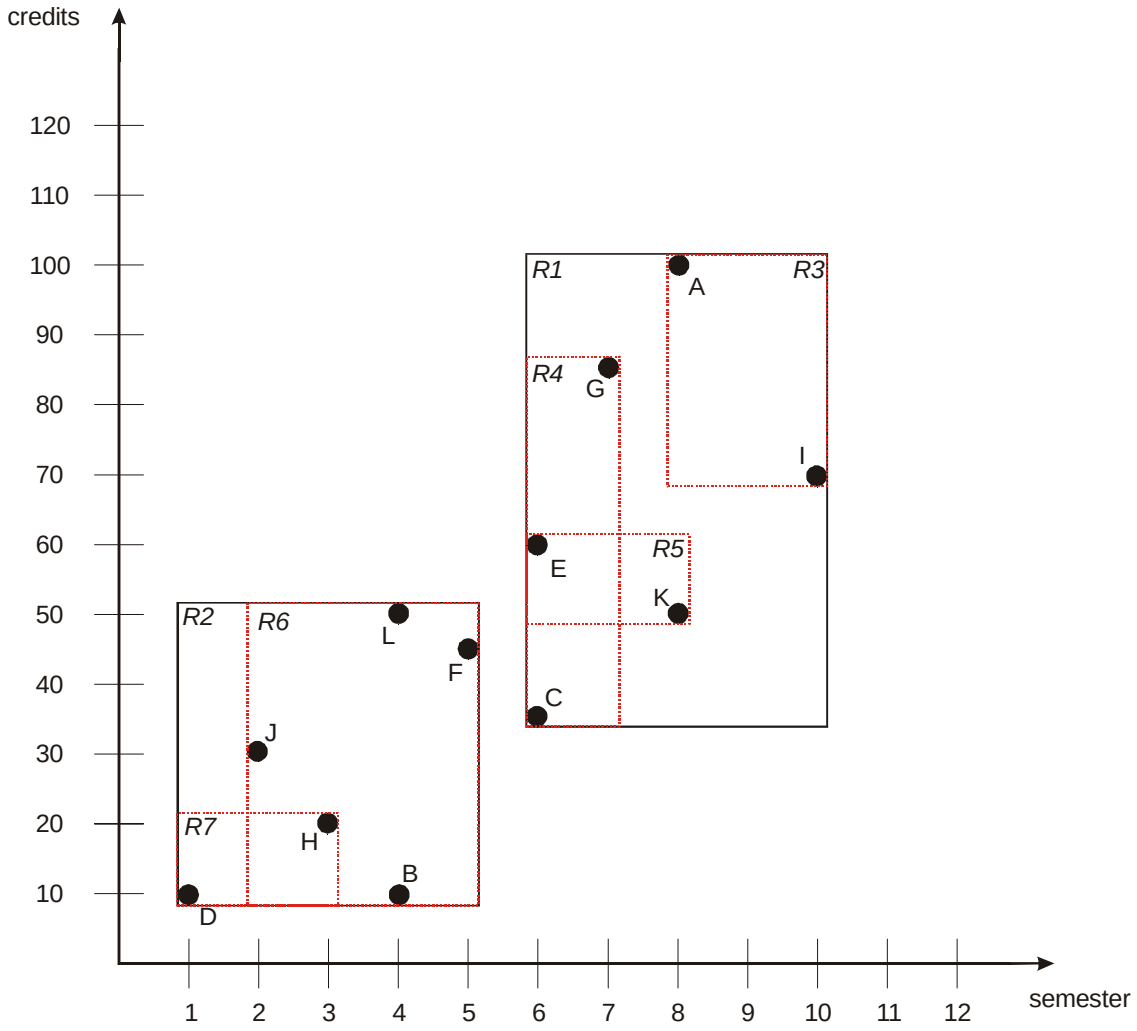

Figure 5: The graphical display of the example database

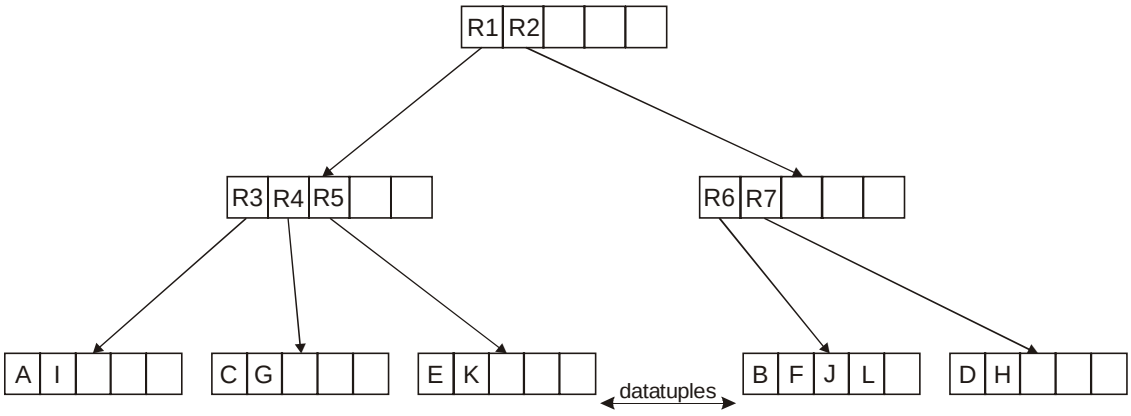

Figure 6: The example database as R-Tree structure

3.1 Searching

The searching in an R-Tree works similar to the search in an B-Tree; the tree will be descended from the root. But since it could happen that in an R-Tree - in distinction to the B-Tree - several rectangles, which have to be searched, are overlapping (see fig. 7). Because all these sub rectangles have to be visited, no good worst-case performance can be guarantied

Algorithm: ***Search***

Let *T* be the root of an R-Tree. We search all index records whose rectangles overlap a specified search rectangle *S*.

- If T is not a leaf, then apply ***Search*** on every child whose root is pointed by *child-pointer* and that overlaps with S.

- If T is a leaf, return all entries which overlap with S as the result set.

An example for *search* according to our example database:

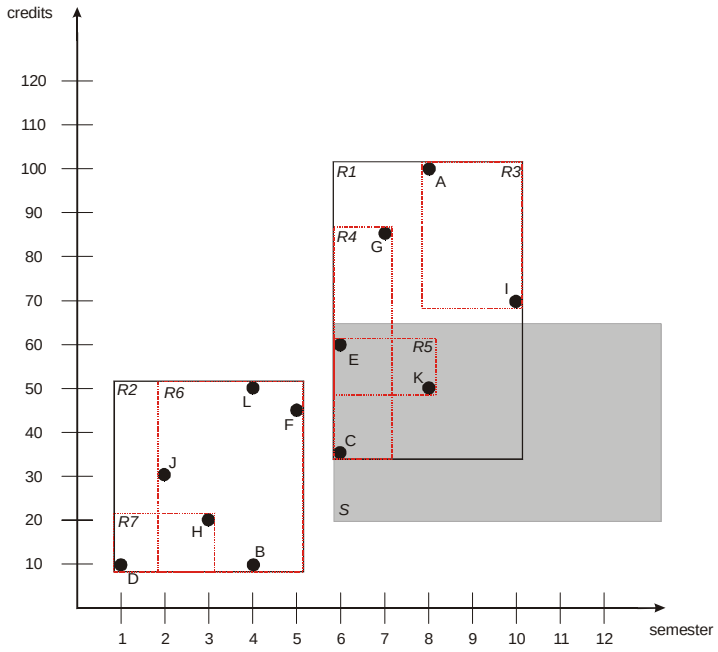

Figure 7: The search rectangle S over the graphical display of the example database

In this example we want to find all students that are in the sixth semester or higher and earned between 20 and 65 credits.
R1 overlaps the query rectangle S, not R2, so we search in R1. In the next step R4 and

R5 are overlapping with S in these rectangles we find the results which are inside the search rectangle. From R4 we get C and from R5 we get E and K, so the result set is {C, E, K}.
Searching for point data works similar, only the query rectangle is just a query point (e.g. (6, 35) will return C as result).

3.2 Insertion

If a new entry has to be inserted into a database, a new index record has to be added to the R-Tree. This is also the only chance for the R-Tree to grow in height, namely if there is a node overflow, the node has to be split. In the case that the split reaches the root, the height will grow . (Note: The splitting is explained in section 3.4.)

Algorithm: ***Insert***

- Let E be the new entry.
- Use ***ChooseLeaf*** to find the leaf node L where E has to be placed in.
- If there is enough space in L (no node overflow), add E to it. Else apply ***SplitNode*** to L, which will return L and L' containing E and all the former elements of L.
- Apply ***AdjustTree*** on L, if there was a split before do this also on L'.
- If the split reaches the root and it has to be split, create a new root whose children are the two nodes which the split of the root returned.

Algorithm: ***ChooseLeaf***

Select a suitable leaf node to place the new entry E in.
Let N be the root node.

- If N is a leaf, return it.
- If N is not a leaf, find the entry F_k in N, whose rectangle has to be enlarged least to include the rectangle of E.
  For the case that several entries F_k are found, choose the smallest one.
- Apply ***ChooseLeaf*** on the chosen F_k until a leaf is reached.

Algorithm: ***AdjustTree***

Climb from a leaf node L to the root, while adjusting the rectangles and doing necessary node splits.
Set N=L. If L was split formerly set N' as L'.

- If N is the root, end.
- Let P be the parent of N. Adjust N's entry in P so that is covers all rectangles in N.
- If a splitting has occurred, add a new entry to P that points to N'. If the parent node P overflows, invoke ***SplitNode*** (see section 3.4) on it.

An example for *Insertion*:

We want to insert a new student (Q, 10, 65). *ChooseLeaf* returns R1 as the first new node. Afterwards R3 is chosen as the rectangle where to insert the entry and it does not overflow. So Q is inserted in R3. After that *AdjustTree* updates the rectangle of R3 and R1.

3.3 Deletion

If an object has to be removed from a database, you must find the according index record E and remove it.
This is also the only chance for the R-Tree to decrease in height (see *Delete* algorithm).

Algorithm: ***Delete***

- Apply ***FindLeaf*** to find the leaf L that contains E. End if E was not found.
- Remove E from L.
- Use ***CondenseTree*** on L to condense under full nodes.
- If the root has only one child after adjusting, let the child become the new root (The height of the Tree decreases.).

Algorithm: ***FindLeaf***

Find the leaf node that contains the index record E. Let T be the root.

- If T is no leaf, apply ***FindLeaf*** on all children whose rectangles overlap the one of E. If E is found return it.

- If T is a leaf, compare each entry to E and if they match return T.

Algorithm: ***CondenseTree***

This algorithm takes a leaf node L from which an entry has been deleted and abolish the node if it has too few (less than *m*) entries.
The algorithm goes upwards through the whole tree and adjusts all rectangles, if necessary, to make them smaller.

Set N=L and Q is the empty set of abolished nodes.

- If N is the root go to the last step. Otherwise let P be the parent of N.

- If N has less than *m* entries (underflow), remove N's entries in P and add them to Q.

- If there was no underflow in N, adjust its rectangle to the *MBR* that contains all remaining entries in N.

- Set N=P and go to the first step.

- Reinsert all leaves stored in Q into the tree by using ***insert***. All non-leaf nodes that are stored in Q must be inserted in a higher level of the tree so that the tree remains height balanced.

Guttman pointed out, that most of the procedures are similar to the ones in the B-Tree but in distinction to the B-Tree, were the nodes are merged, the nodes in the R-Tree are reinserted instead. That is of comparable efficiency but much easier to implement because the already existing algorithm *insert* can be used.
In an R-Tree, it is also supported to delete all data objects in a certain area.

An example for a *deletion*:

We want to delete student K from our database. Applying *FindLeaf* returns R5 as the corresponding rectangle, we remove the entry K from R5 and notice that R5 underflows. Afterwards we apply *CondenseTree* on R5, which will remove R5 from the

R-Tree, add E to R4 by using *Insert* and update the rectangle of R1.

3.4 Splitting a Node

When adding a new entry to a full node (a node containing M entries), it is necessary to split these M+1 entries up in two new nodes.
When thinking about how to divide them, the goal should be to minimize the size of the two resulting rectangles because the smaller they are the better is the chance for the search algorithm to visit only the nodes that are necessary to visit. It's because the smaller the rectangles are the smaller is the possibility that they overlap with others.
In figure 8a you can see an example for a good split and in figure 8b one for a bad split.

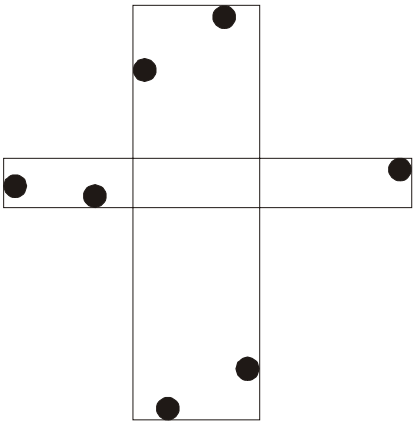

Figure 8a: A good split

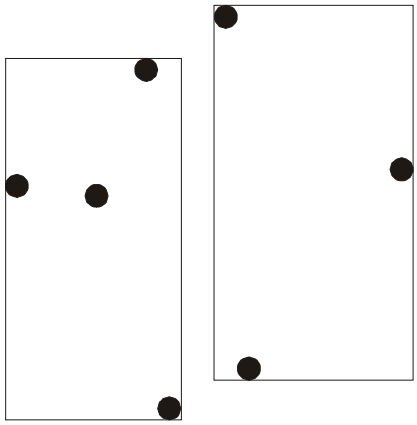

Figure 8b: A bad split

There are three algorithms ***SplitNode***, offered by Guttman, to divide M+1 entries in two nodes N and N'. Their names tell us how their CPU usage increases relative to M.

Exhaustive splitnode algorithm:

This algorithm goes straight through all possibilities of grouping and chooses the best.
It has the best quality but since there are 2^{M-1} possibilities of combinations the CPU usage grows exponential. So it is not very appropriate for larger database where the *M* is very high (>50).

Quadratic cost algorithm:

This algorithm picks the two entries that are most distant from each other (which means that their bounding box would need the most space) and put them into different nodes.

All remaining elements will be distributed considering to the same pattern. For each node the required space when adding it to N or N' is calculated. Then the node with the greatest difference between these two groups will be added to the node (which means that it will be put into the rectangle of a node from the upper level) requiring less space as if added it to another one.
The quadratic cost algorithm does not produce the best splits but is more efficient for large databases which contain more than 50 entries.

Two other methods are used for this algorithm:
PickSeeds: Selects the two first elements of the groups.
PickNext: Selects one of the remaining entries to put in a group.

Linear cost algorithm:

This one chooses for each dimension $i \in \{0, \ldots, k-1\}$ the two entries that are the most distant ones from each other in this dimension i and puts them into different nodes. After that has been done for every dimension, the left nodes are randomly distributed.
The dissimilarity to the quadratic cost algorithm is localized in the methods *PickSeeds* and *PickNext*.
For the reason that this algorithm is very speedy, it can outweigh his bad search performance.

4 Derivatives of the R-Tree

After the development of the original R-Tree there had been some improvements and special properties added to the original structure and the algorithms, which can be seen below.

The packed R-Tree (1985): Space that is not used by the index structure of the Tree will be freed. It is often used in databases where nothing has to be deleted or inserted.

The R^+-Tree (1987): A new insertion method tries to avoid overlapping regions, which results in higher costs for disk utilisation but also in a faster search [BKSS90].

The R^*-Tree (1990): The structure is equal to the R-Tree. The difference to the R-Tree is that some improved insert and split methods had been found by researching and testing.

The X-Tree (1996): The latest development of an index structure for spatial data. By varying the node size overlapping regions are avoided [HR99].

5 Conclusion

As a conclusion we say that the R-Tree in the original form was already well suited to handle spatial data objects that means it is a useful index structure for multidimensional data. Though the derivatives have some improvements which let them become more suitable for special situations.
It is easy to implement an R-Tree to database systems that support conventional access methods like the B-Tree. However the R-Tree is not very well suited for point data because of the search method which does not guarantee that only one search path is required.
But it was not possible to find any test results which compare the R-Tree to other tree structure like the B-Tree, except a comparison of the R*-Tree to the GRID-File in [BKSS90].
The structure of the R-Tree does not differ much from the one of the B-Tree, only when deleting objects and you have to divide the rest of a node over the tree, it is not necessary to merge nodes because in the R-Tree these entries will simply be inserted with the normal ***insert*** method (see sec. 3.2). All leaves have to appear on the same level which means the tree has to be kept height balanced.

6 References

[BKSS90] N.Beckmann, H.-P. Kriegel, R. Schneider and B. Seeger. The R*-Tree: an efficient and robust access method for points and rectangles. ACM SIGMOD, pages 322-331, May 1990.

[Gut84] A. Guttman. R-Trees: A Dynamic Structure for Spatial Searching. In: SIGMOD 84, Proceedings of Annual Meeting, pages 47-57. Boston, 1984.

[HR99] T. Haerder und E. Rahm: Datenbanksysteme -- Konzepte und Techniken der Implementierung. Springer Verlag, 1999.

[SRF87] Timos K. Sellis, Nick Roussopoulos, Christos Faloutsos: The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. VLDB 1987: 507-518