Database Technology

Indexing with R-trees

Motivation

In many real-life applications objects are represented as multidimensional points:

Spatial databases

(e.g., points in 2D or 3D space)

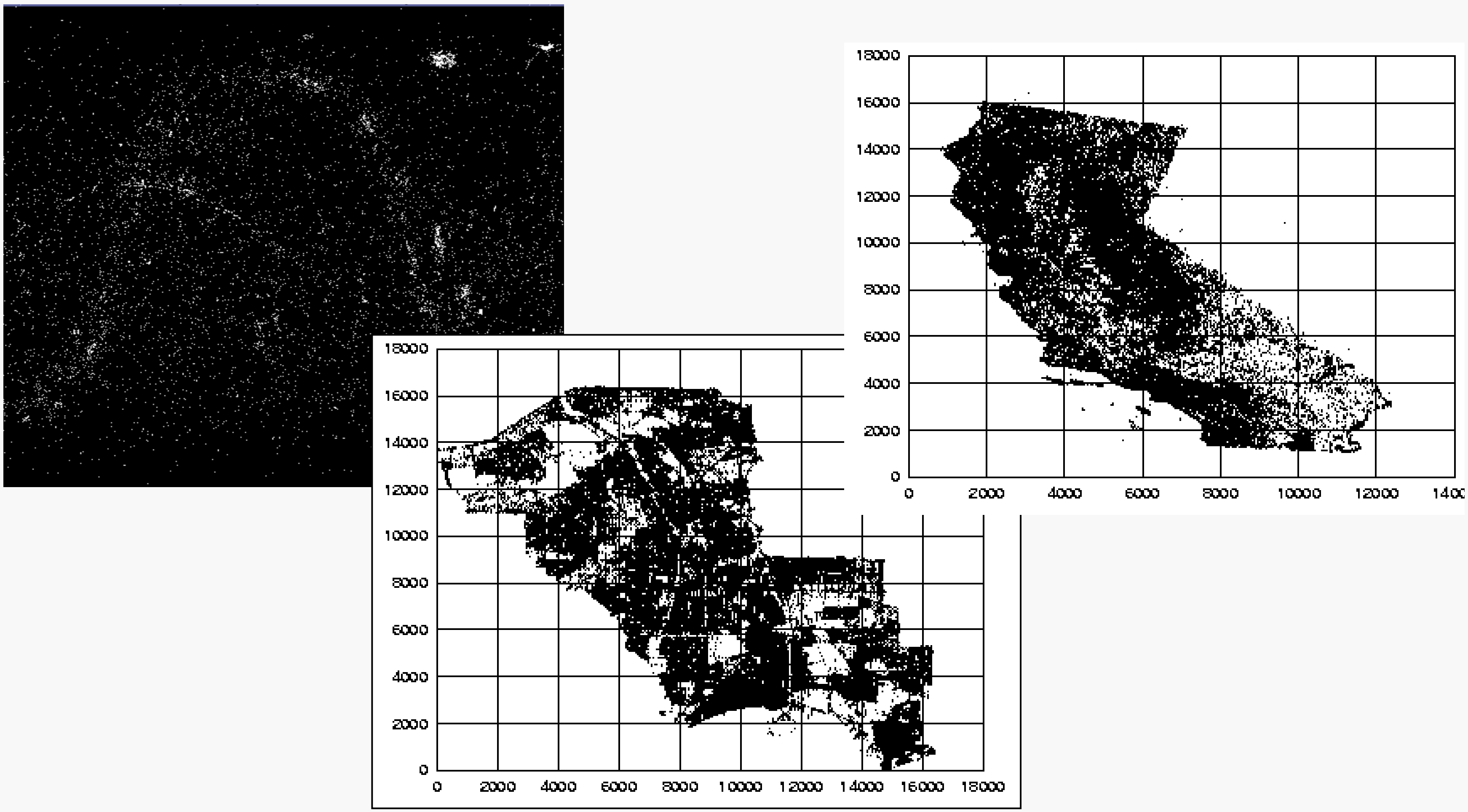
Multidimensional databases

(each record is considered a point in *n*-D space)

Multimedia databases

(e.g., feature vectors are extracted from images, audio files)

Examples of 2D Data Sets



Requirements

- Indexing scheme are needed to speed up query processing.
- We need disk-based techniques, since we do not want to be constrained by the memory capacity.
- The methods should handle insertions/deletions of objects (i.e., they should work in a dynamic environment).

The R-tree

A. Guttman:

"R-tree: A Dynamic Index Structure for Spatial Searching",

ACM SIGMOD Conference, 1984

R-tree Index Structure

- An R-tree is a height-balanced tree similar to a B-tree.
- Index records in its leaf nodes containing pointers to data objects.
- Nodes correspond to disk pages if the index is disk-resident.
- The index is completely dynamic.
- Leaf node structure (*r*, *objectID*)
  - *r*: minimum bounding rectangle of object
  - *objectID*: the identifier of the corresponding object
- Nonleaf node structure (*R*, *childPTR*)
  - *R*: covers all rectangles in the lower node
  - *childPTR*: the address of a lower node in the R-tree

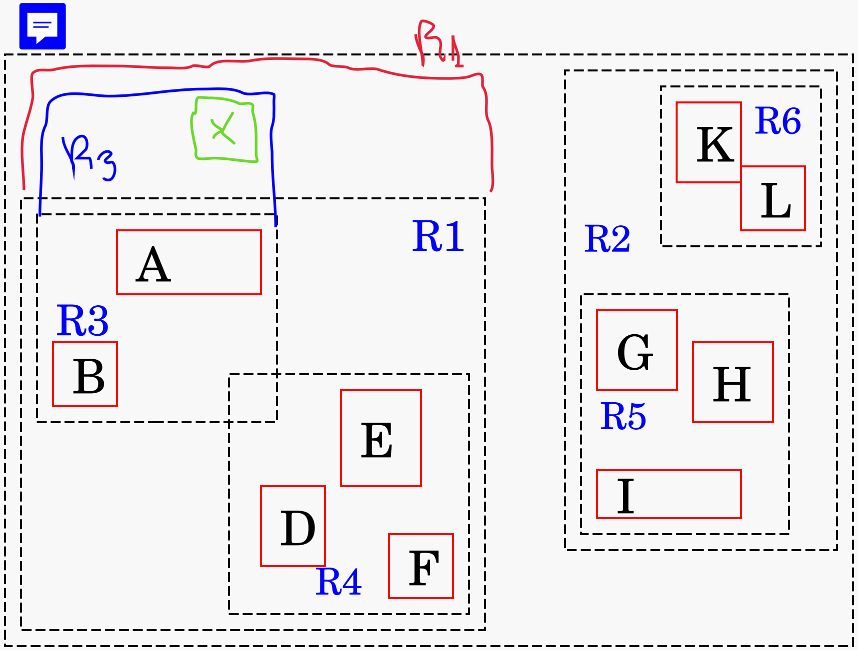
Properties of the R-tree

M: maximum number of entries that will fit in one node

m: minimum number of entries in a node, $m \leq M/2$

- Every leaf node contains between b and M index records unless it is the root.
- For each index record (*r*, *objectID*) in a leaf node, *r* is the smallest rectangle (Minimum Bounding Rectangle (MBR)) that spatially contains the data object.
- Every non-leaf node has between *m* and *M* children, unless it is the root.
- For each entry (*R*, *childPTR*) in a non-leaf node, *R* is the smallest rectangle that spatially contains the rectangles in the child node.
- The root node has at least 2 children unless it is a leaf.
- All leaves appear at the same level.

R-Tree Example



M : maximum number of entries

m : minimum number of entries (≤ M/2)

(1) Every leaf node contains between *m* and *M* index records unless it is the root.

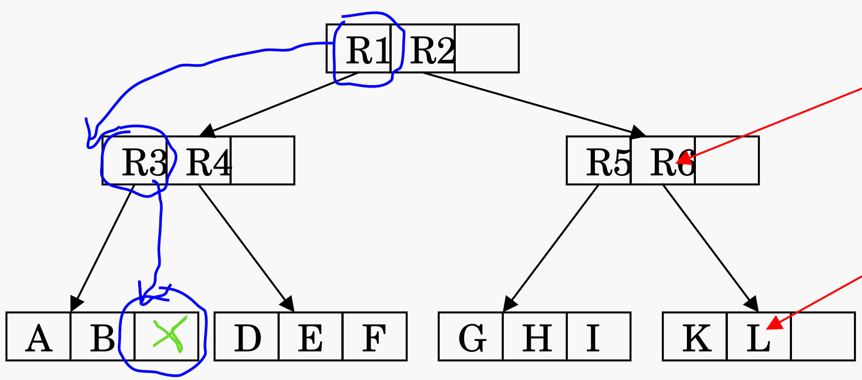
(2) Each leaf node has the smallest rectangle that spatially contains the n-dimensional data objects.

(3) Every non-leaf node has between *m* and *M* children unless it is the root.

(4) Each non-leaf node has the smallest rectangle that spatially contains the rectangles in the child node.

(5) The root node has at least two children unless it is a leaf.

(6) All leaves appear on the same level.



<MBR, Pointer to a child node>

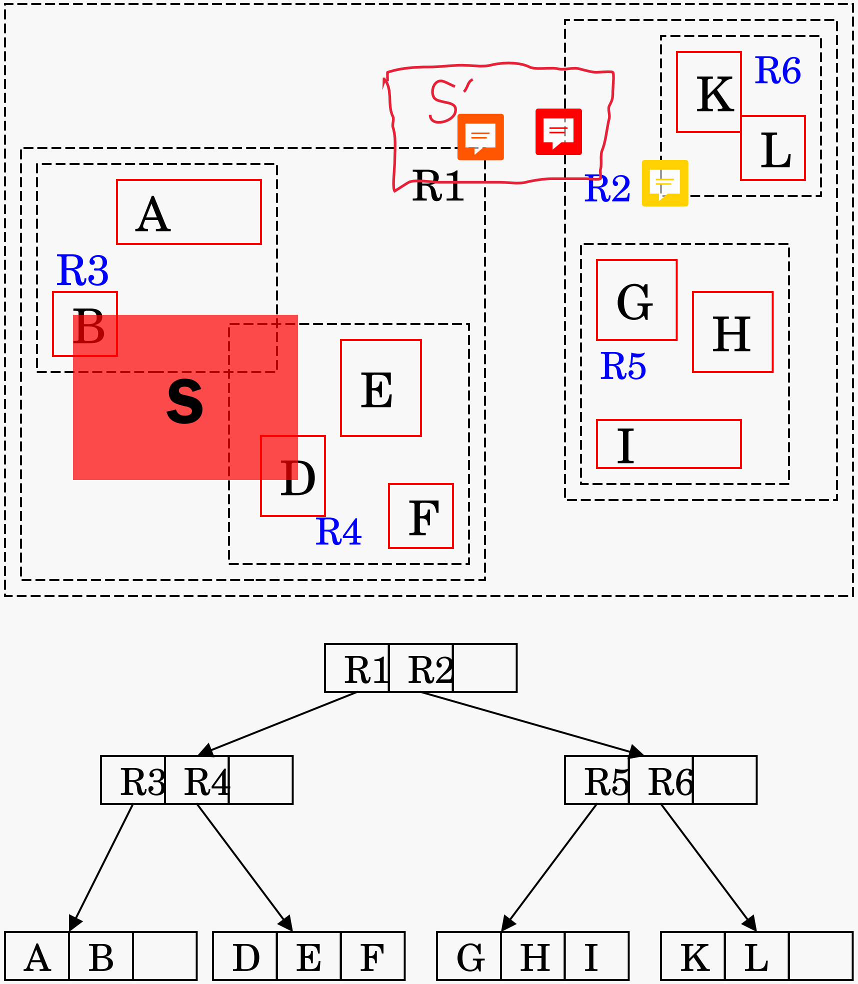
<MBR, Pointer or ID>

R-tree Search Algorithm

- Given an R-tree whose root is *T*, find all index records whose rectangles overlap a search rectangle *S*.

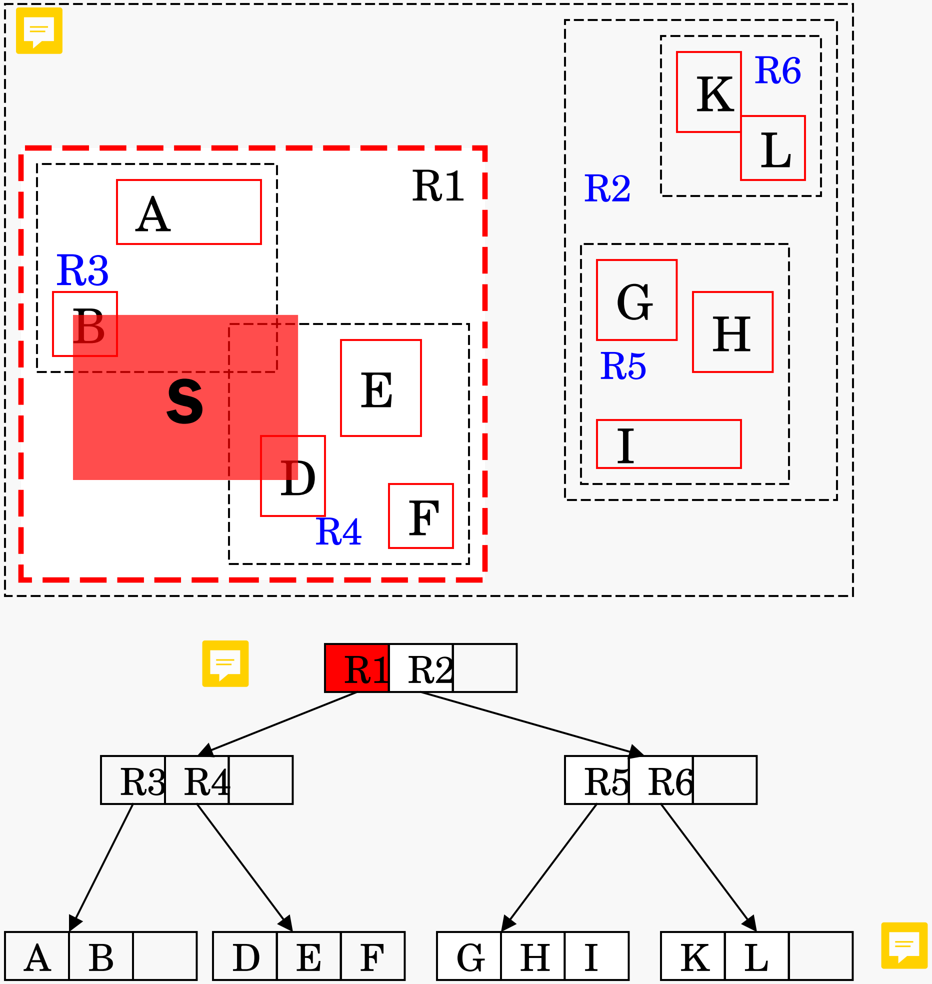
- Algorithm <u>Search</u>
  - [Search subtrees]
    - If *T* is not a leaf, check each entry *E* to determine whether *E*.*R* overlaps *S*.
    - For all overlapping entries, invoke <u>Search</u> on the tree whose root is pointed to by *E*.*childPTR*.
  - [Search leaf node]
    - If *T* is a leaf, check all entries *E* to determine whether *E*.*r* overlaps *S*. If so, *E* is a qualifying record.

R-Tree Search Example (1/7)

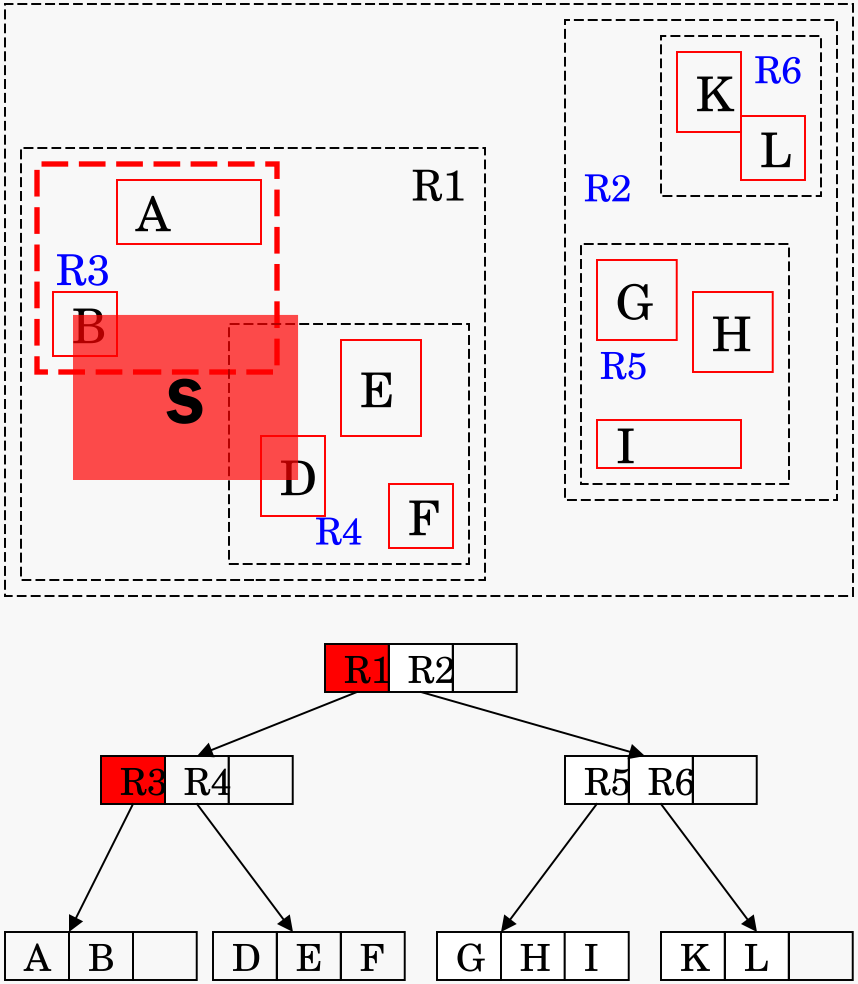


Find all objects whose rectangles are overlapped with a search rectangle S

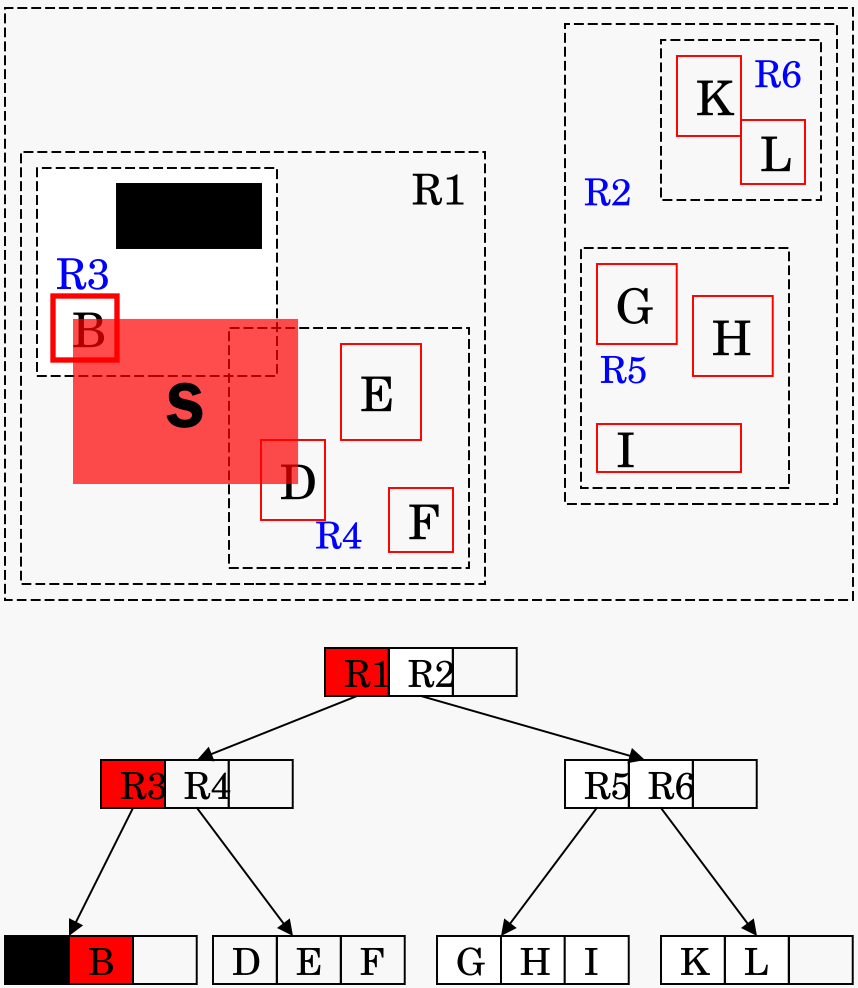
R-Tree Search Example (2/7)



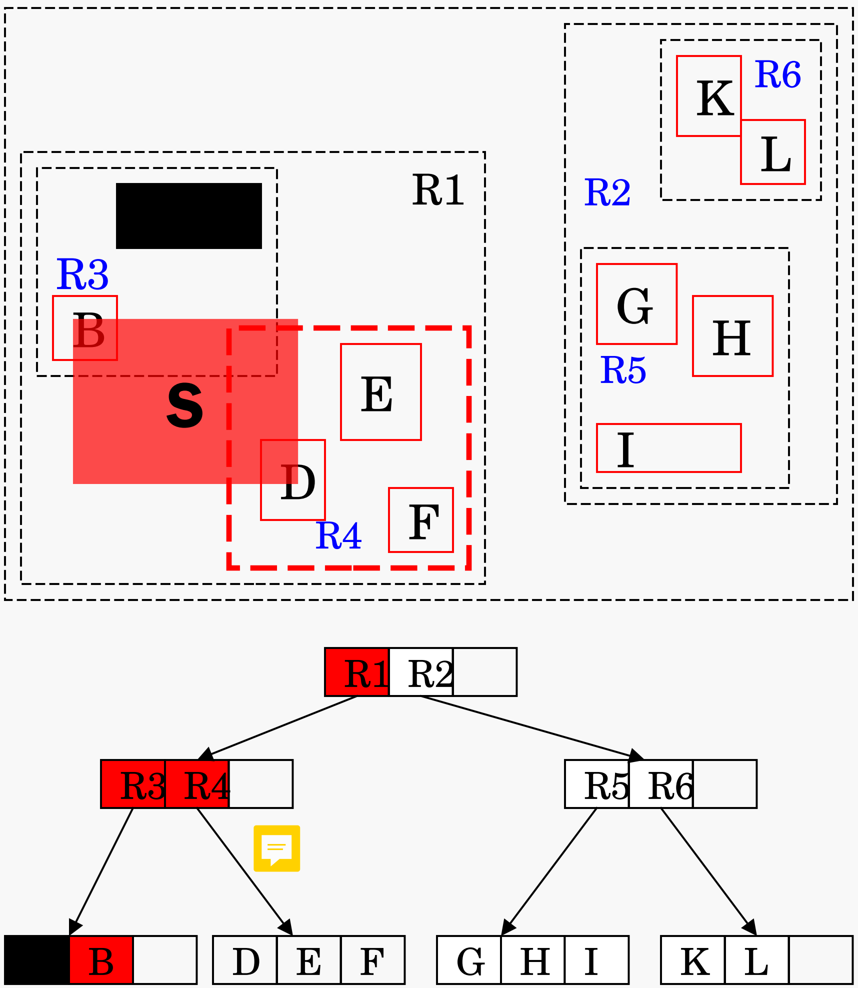
R-Tree Search Example (3/7)



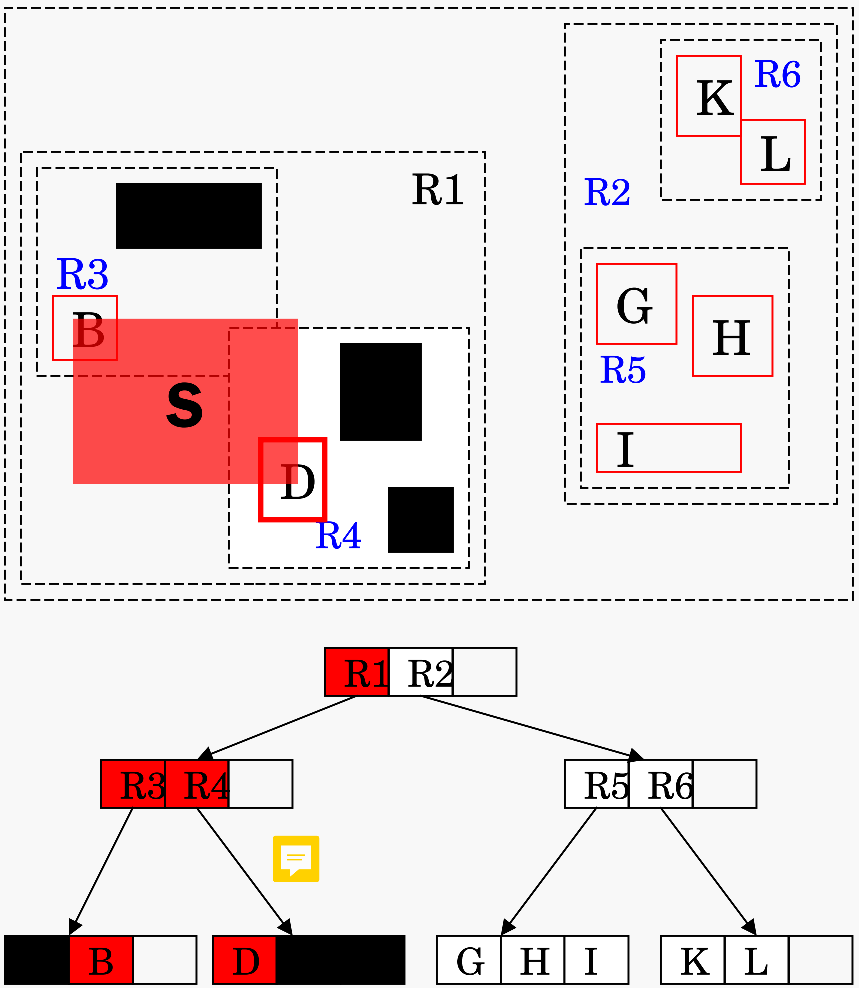
R-Tree Search Example (4/7)



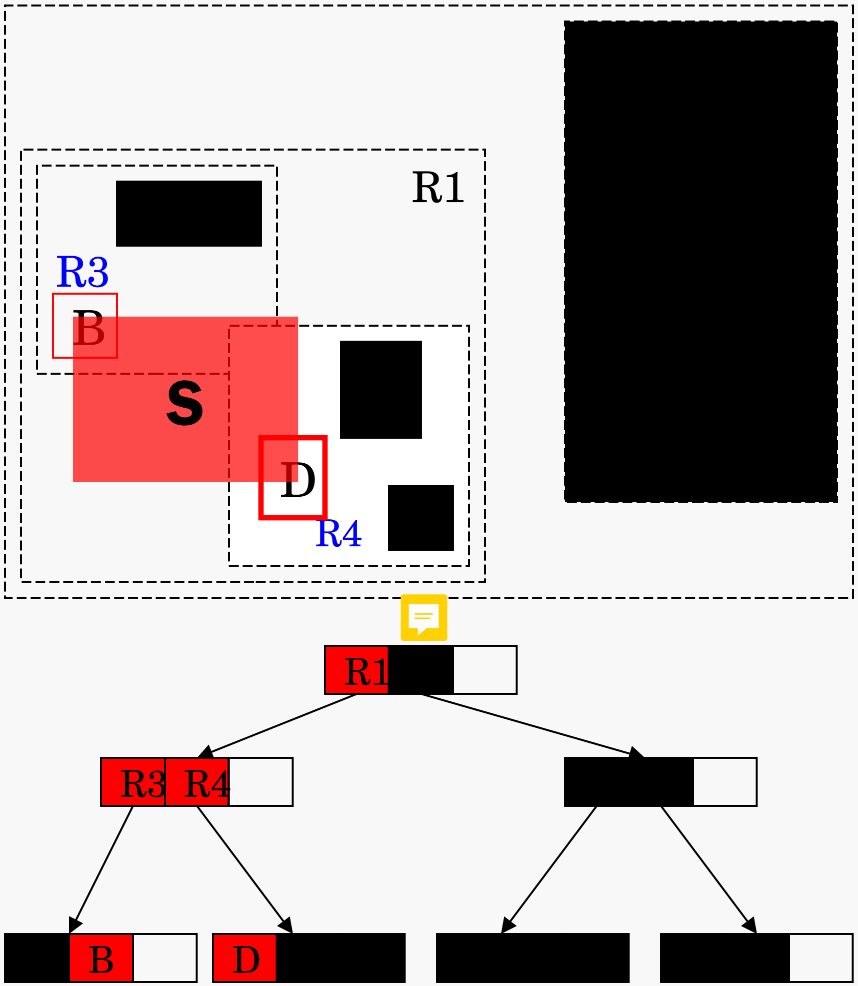
R-Tree Search Example (5/7)



R-Tree Search Example (6/7)



R-Tree Search Example (7/7)



B and D
→ overlapped objects with S

R-tree Insertion

- Algorithm Insert
  // Insert a new index entry E into an R-tree.
  - (1) [Find position for new record]
    - Invoke ChooseLeaf to select a leaf node L in which to place E.
  - (2) [Add record to leaf node]
    - If L has room for another entry, install E.
    - Otherwise invoke SplitNode to obtain L and LL containing E and all the old entries of L.
  - (3) [Propagate changes upward]
    - Invoke AdjustTree on L, also passing LL if a split was performed
  - [Grow tree taller]
    - If node split propagation caused the root to split, create a new root whose children are the 2 resulting nodes.

Algorithm ChooseLeaf (1)

// Select a leaf node in which to place a new index entry E.

- [Initialize]
  - Set N to be the root node.
- [Leaf check]*
  - If N is not a leaf, return N.
- [Choose subtree]
  - If N is not a leaf, let F be the entry in N whose rectangle F.I needs least enlargement to include E.I.

    Resolve ties by choosing the entry with the rectangle of smallest area.
- [Descend until a leaf is reached]
  - Set N to be the child node pointed to by F.p.
  - Repeat from *.

Algorithm AdjustTree (3)

// Ascend from a leaf node L to the root, adjusting covering rectangles and propagating node splits as necessary.

- [Initialize]
  - Set N=L. If L was split previously, set NN to be the resulting second node.
- [Check if done]*
  - If N is the root, stop.
- [Adjust covering rectangle in parent entry]
  - Let P be the parent node of N, and let E_n be N's entry in P.
  - Adjust E_n.I so that it tightly encloses all entry rectangles in N.
- [Propagate node split upward]
  - If N has a partner NN resulting from an earlier split, create a new entry E_{NN} with E_{NN}.p pointing to NN and E_{NN}.I enclosing all rectangles in NN.
  - Add E_{NN} to P if there is room. Otherwise, invoke SplitNode to produce P and PP containing E_{NN} and all P's old entries.
- [Move up to next level]
  - Set N = P and set NN = PP if a split occurred. Repeat from *.

R-tree Deletion

- Algorithm Delete
  // Remove index record E from an R-tree.
  - [Find node containing record]
    - Invoke FindLeaf to locate the leaf node L containing E.
    - Stop if the record was not found.
  - [Delete record]
    - Remove E from L.
  - [Propagate changes]
    - Invoke CondenseTree, passing L.
  - [Shorten tree]
    - If the root node has only one child after the tree has been adjusted, make the child the new root.

- Algorithm FindLeaf
  // Find the leaf node containing the entry E in an R-tree with root T.
  - [Search subtrees]
    - If T is not a leaf, check each entry F in T to determine if F.I overlaps E.I. For each such entry invoke FindLeaf on the tree whose root is pointed to by F.p until E is found or all entries have been checked.
  - [Search leaf node for record]
    - If T is a leaf, check each entry to see if it matches E. If E is found return T.

Algorithm CondenseTree

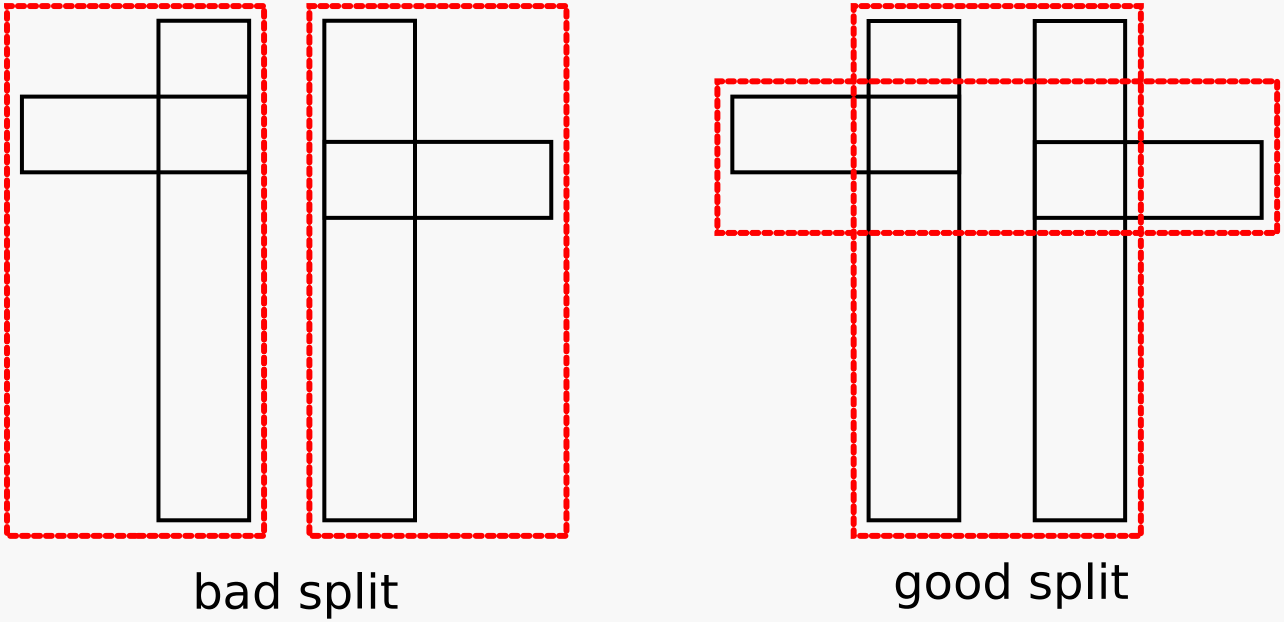
// Given a leaf node L from which an entry has been deleted, eliminate the node if it
// has too few entries and relocate its entries.
// Propagate node elimination upward as necessary.
// Adjust all covering rectangles on the path to the root.

- [Initialize]
  - Set N=L. Set Q, the set of eliminated nodes, to be empty.
- [Find parent entry]*
  - If N is the root, go to +.
  - Otherwise, let P be the parent of N, and let E_n be N's entry in P.
- [Eliminate under-full node]
  - If N has fewer than m entries, delete E_n from P and add N to set Q.
- [Adjust covering rectangle]
  - If N has not been eliminated, adjust E_n.I to tightly contain all entries in N.
- [Move up one level in tree]
  - Set N=P and repeat from *.
- [Re-insert orphaned entries]+
  - Re-insert all entries of nodes in set Q.

    Entries from eliminated leaf nodes are re-inserted in tree leaves as described in Insert, but entries from higher-level nodes must be placed higher in the tree, so that leaves of their dependent subtrees will be on the same level as leaves of the main tree.

Node Splitting

- The total area of the 2 covering rectangles after a split should be minimized.
  - ⇨ The same criterion was used in ChooseLeaf to decide a new index entry: at each level in the tree, the subtree chosen was the one whose covering rectangle would have to be enlarged least.



Node Split Algorithms

- Exhaustive Algorithm
  - To generate all possible groupings and choose the best.
    - ⇨ The number of possible splits is very large.
- Quadratic-Cost Algorithm
  - Attempts to find a small-area split, but is not guaranteed to find one with the smallest area possible.
  - Quadratic in M (node capacity) and linear in dimensionality
  - Picks two of the M+1 entries to be the first elements of the 2 new groups by choosing the pair that would waste the most area if both were put in the same group, i.e., the area of a rectangle covering both entries would be greatest.
  - The remaining entries are then assigned to groups one at a time.
  - At each step the area expansion required to add each remaining entry to each group is calculated, and the entry assigned is the one showing the greatest difference between the 2 groups.

Algorithm Quadratic Split

// Divide a set of M+1 index entries into 2 groups.

- [Pick first entry for each group]
  - Apply algorithm PickSeeds to choose 2 entries to be the first elements of the groups.
  - Assign each to a group.
- [Check if done]*
  - If all entries have been assigned, stop.
  - If one group has so few entries that all the rest must be assigned to it in order for it to have the minimum number m, assign them and stop.
- [Select entry to assign]
  - Invoke algorithm PickNext to choose the next entry to assign.
  - Add it to the group whose covering rectangle will have to be enlarged least to accommodate it.
  - Resolve ties by adding the entry to the group with smaller entry, then to the one with fewer entries, then to either.
  - Repeat from *.

Algorithms PickSeeds & PickNext

- **Algorithm PickSeeds**

  // Select 2 entries to be the first elements of the groups.

  – [Calculate inefficiency of grouping entries together]
    - For each pair of entries E_1 and E_2, compose a rectangle J including E_1.I and E_2.I.
    - Calculate d = area(J) – area(E_1.I) – area(E_2.I).

  – [Choose the most wasteful pair.]
    - Choose the pair with the largest d.

- **Algorithm PickNext**

  // Select one remaining entry for classification in a group.

  – [Determine cost of putting each entry in each group]
    - For each entry E not yet in a group,

      - Calculate d_1 = the area increase required in the covering rectangle of Group 1 to include E.I.

      - Calculate d_2 similarly for Group 2.

  – [Find entry with greatest preference for one group]
    - Choose any entry with the maximum difference between d_1 & d_2.

A Linear-Cost Algorithm

- Linear in M and in dimensionality
- Linear Split is identical to Quadratic Split but uses a different PickSeeds. PickNext simply chooses any of the remaining entries.

- Algorithm LinearPickSeeds

  // Select 2 entries to be the first elements of the groups.
  - [Find extreme rectangles along all dimensions]
    - Along each dimension, find the entry whose rectangle has the highest low side, and the one with the lowest high side.
    - Record the separation.
  - [Adjust for shape of the rectangle cluster]
    - Normalize the separations by dividing by the width of the entire set along the corresponding dimension.
  - [Select the most extreme pair]
    - Choose the pair with the greatest normalized separation along any dimension.

Performance (Insert/Delete/Search)

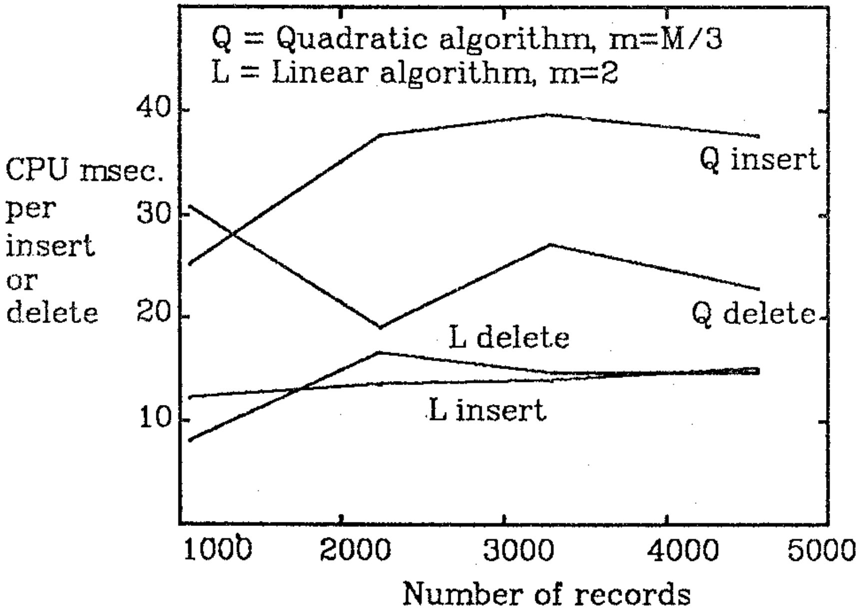


Figure 4.7
CPU cost of inserts and deletes vs. amount of data.

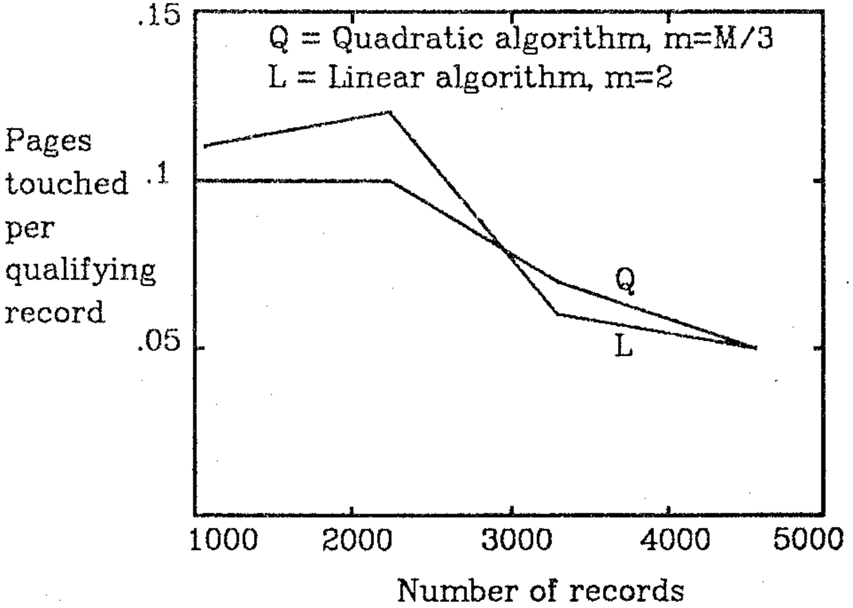


Figure 4.8
Search performance vs. amount of data: Pages touched

Performance (Search/Space)

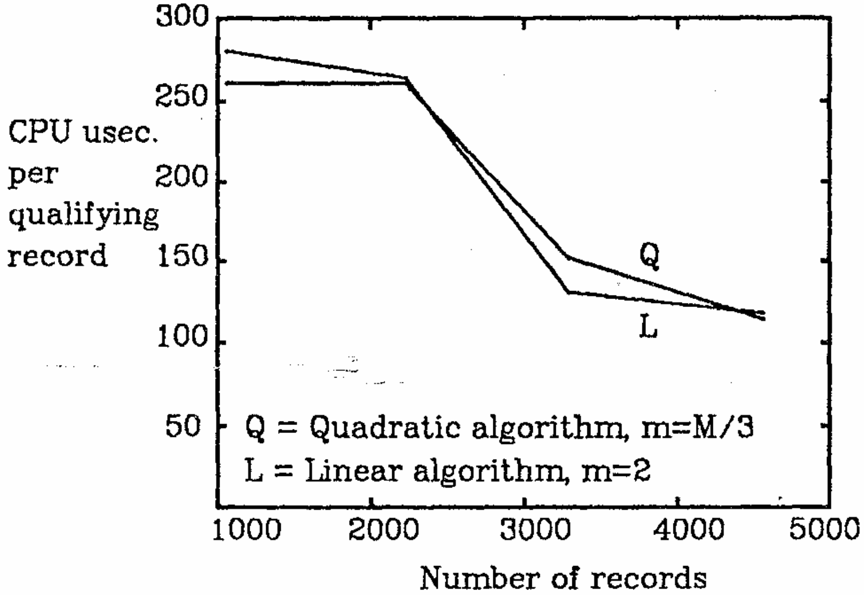


Figure 4.9
Search performance vs. amount of data:
CPU cost

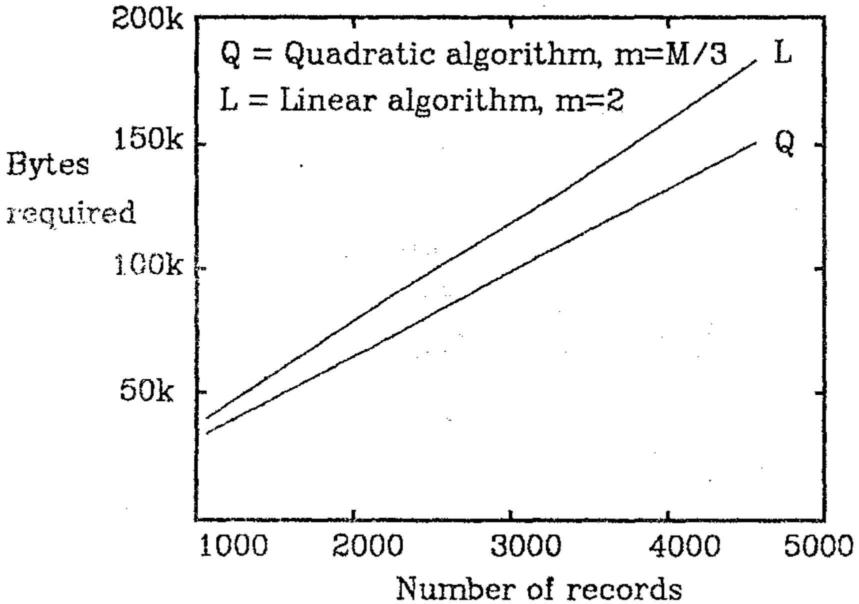


Figure 4.10
Space required for R-tree
vs. amount of data.

Conclusions

- The R-tree structure has been shown to be useful for indexing spatial data objects of non-zero size.
- The linear node-split algorithm proved to be as good as more expensive techniques.

  ⇨ It was fast, and the slightly worse quality of the splits did not affect search performance noticeably.