Department of Informatics
Aristotle University of Thessaloniki
Fall 2016-2017

Multimedia Database Systems

Indexing Part A

Multidimensional Indexing Techniques

Outline

- Motivation
- Multidimensional indexing
  - R-tree
  - R*-tree
  - X-tree
  - Pyramid technique
- Processing Nearest-Neighbor queries
- Conclusions

Motivation

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

Spatial databases

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

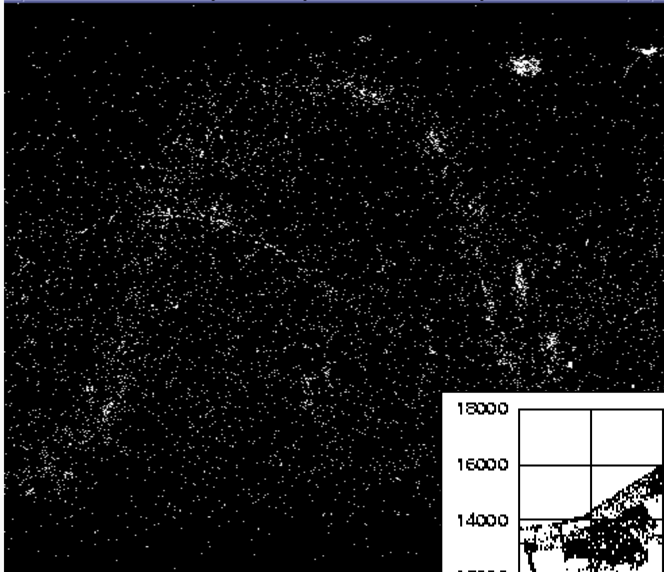
Multidimensional databases

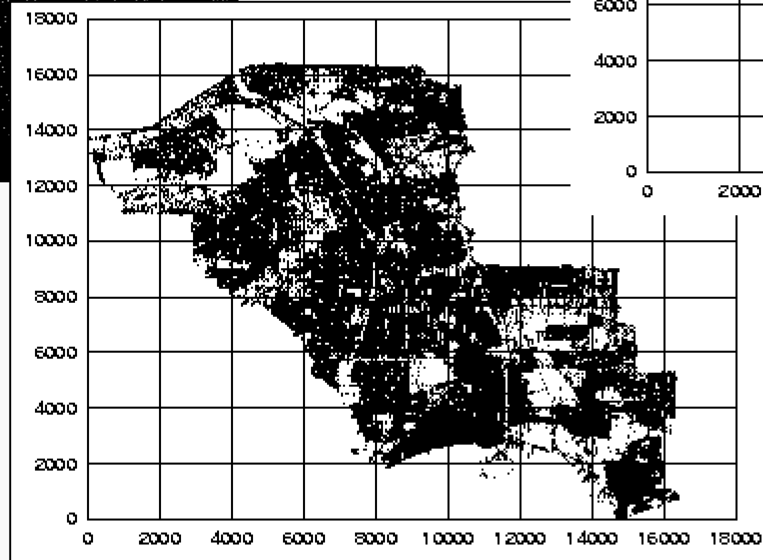
(each record is considered a point in *n*-D space)

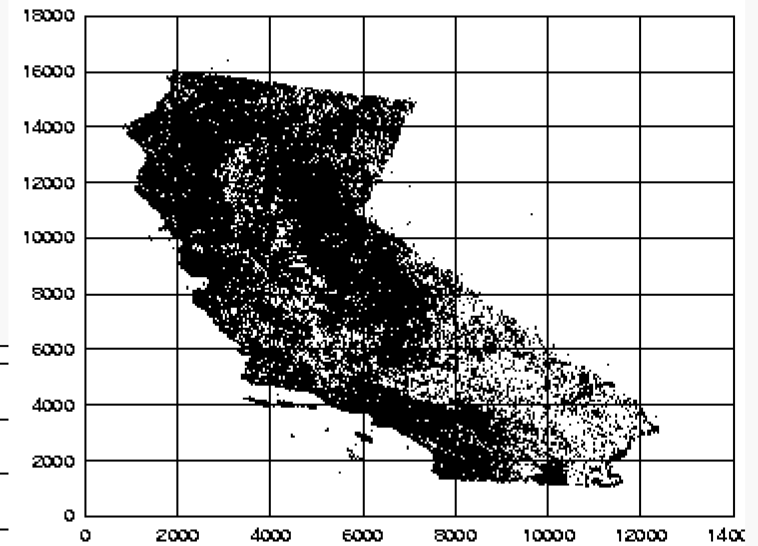
Multimedia databases

(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

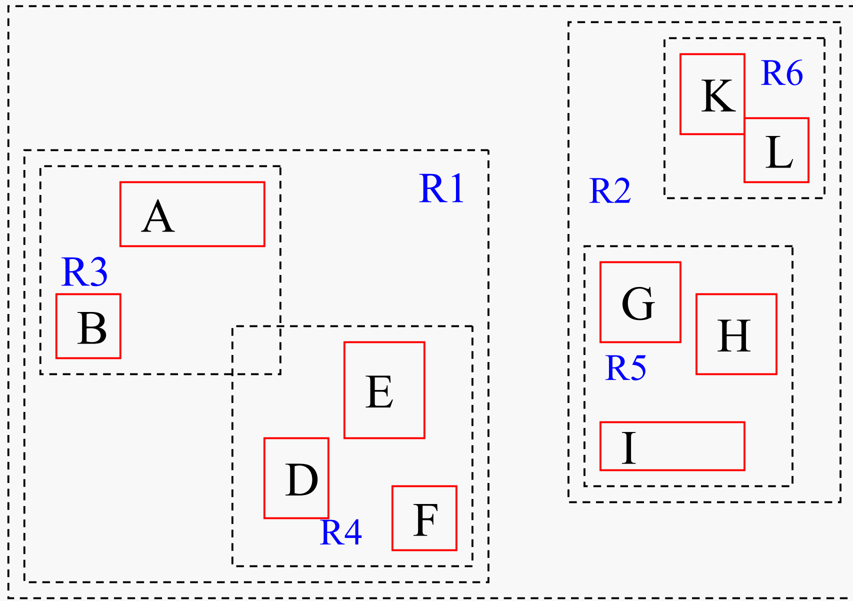
Properties of the R-tree

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

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

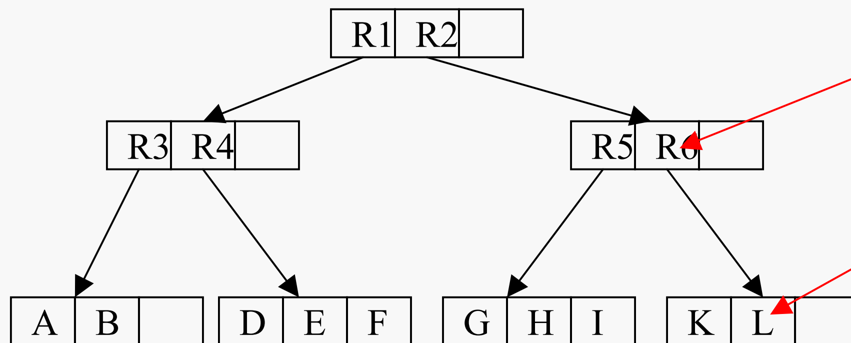
- Every leaf node contains between b and M index records unless it is the root.
- For each index record (r, *objectID*) in a leaf node, r is the smallest rectangle (Minimum Bounding Rectangle (MBR)) that spatially contains the data object.
- Every non-leaf node has between m and M children, unless it 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>

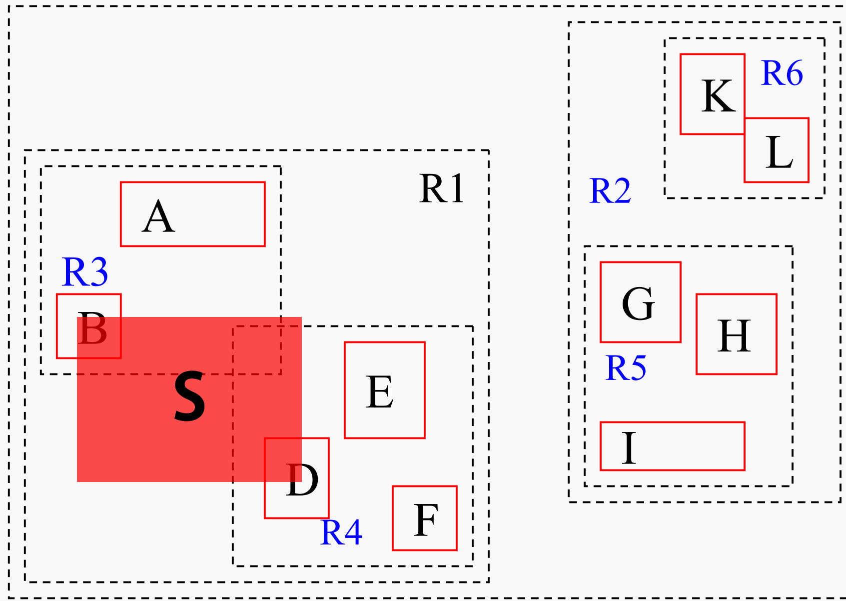
<MBR, Pointer or ID>

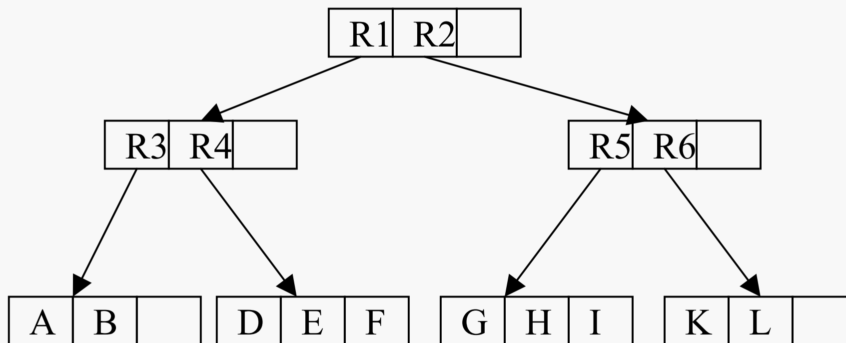
R-tree Search Algorithm

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

- Algorithm Search
  - [Search subtrees]
    - If *T* is not a leaf, check each entry *E* to determine whether *E*.*R* overlaps *S*.
    - For all overlapping entries, invoke Search 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.

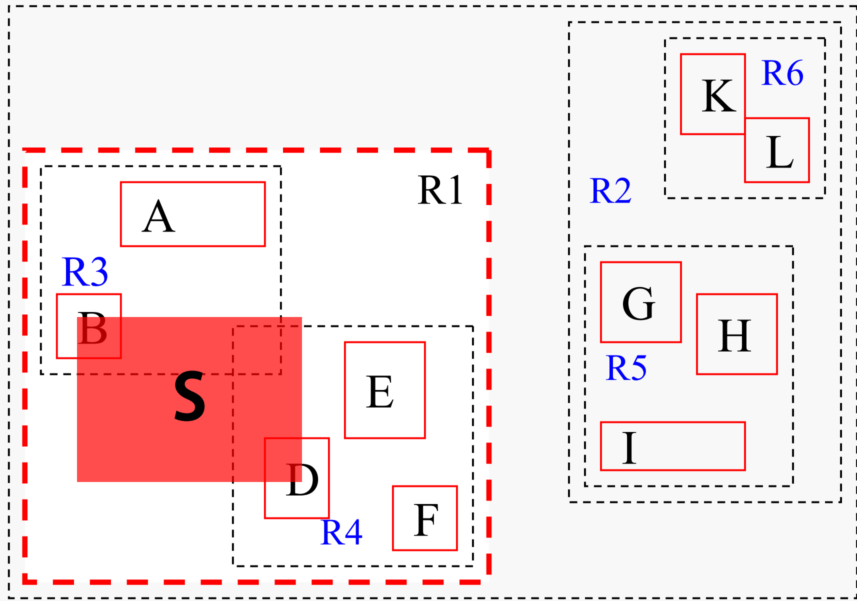
R-Tree Search Example (1/7)

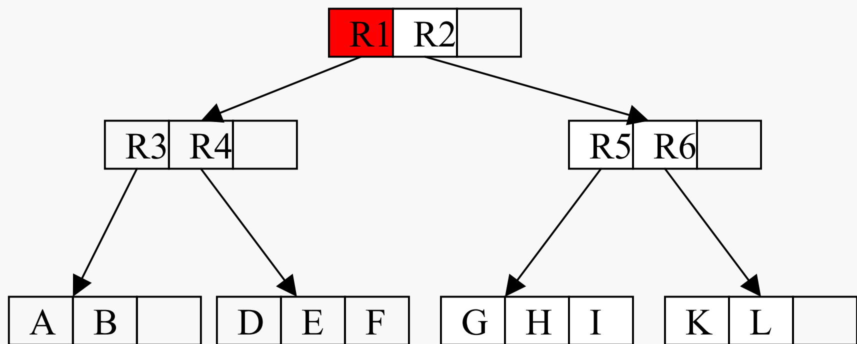




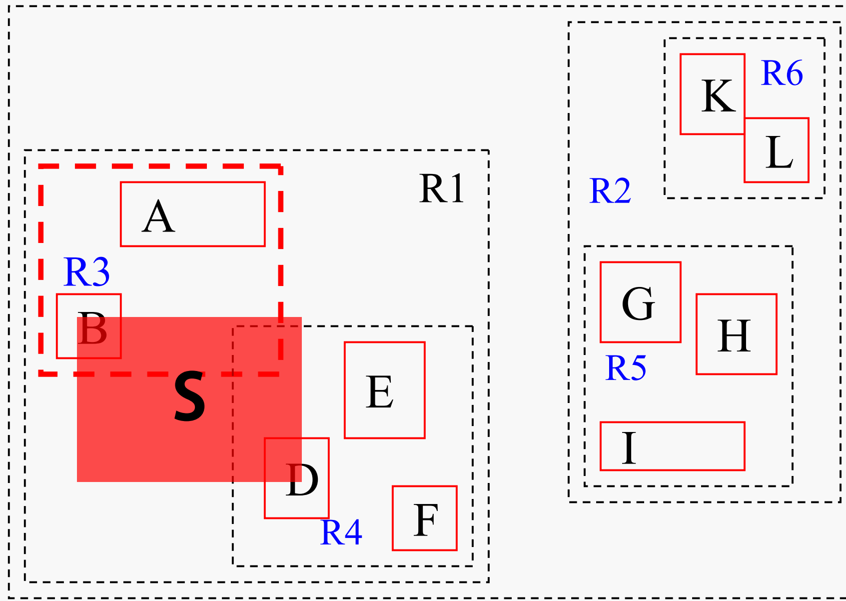
Find all objects whose rectangles are overlapped with a search rectangle S

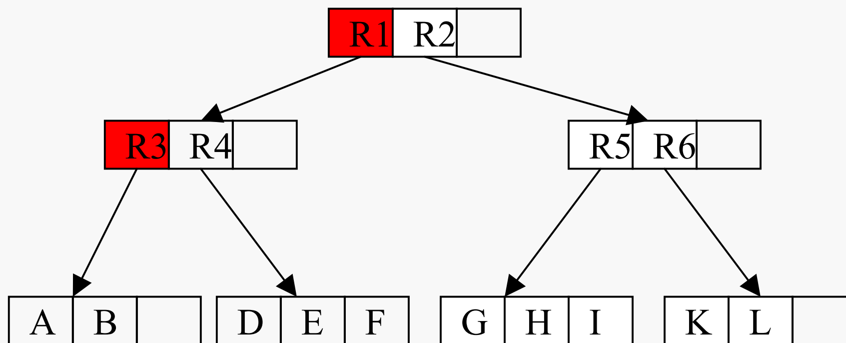
R-Tree Search Example (2/7)



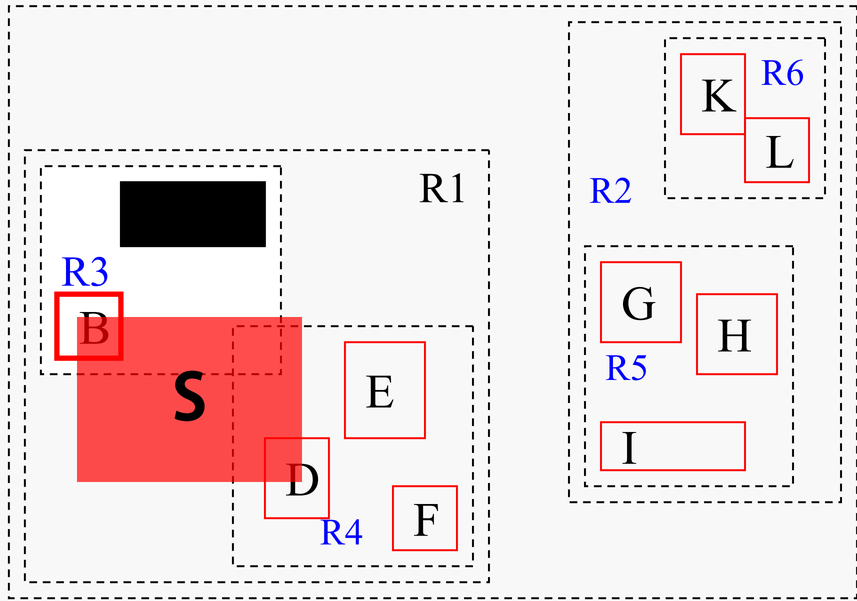


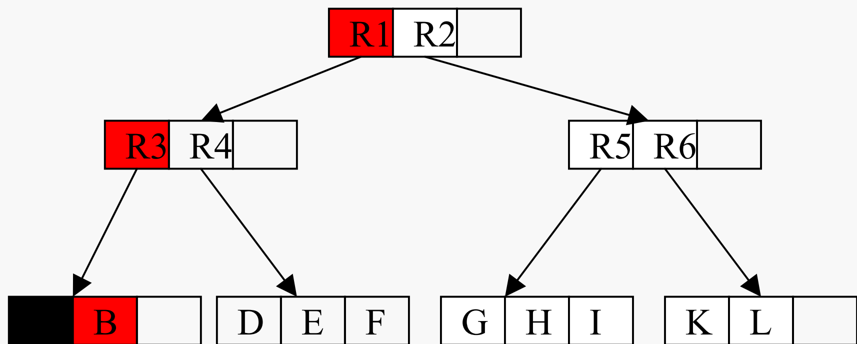
R-Tree Search Example (3/7)



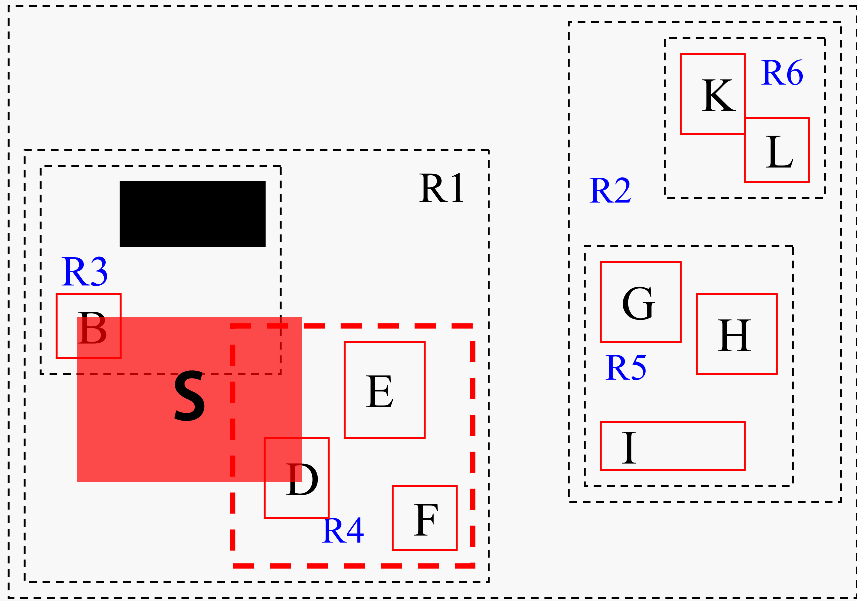


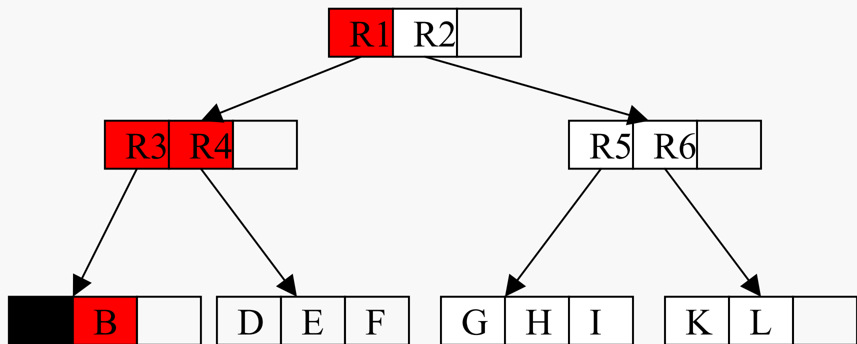
R-Tree Search Example (4/7)



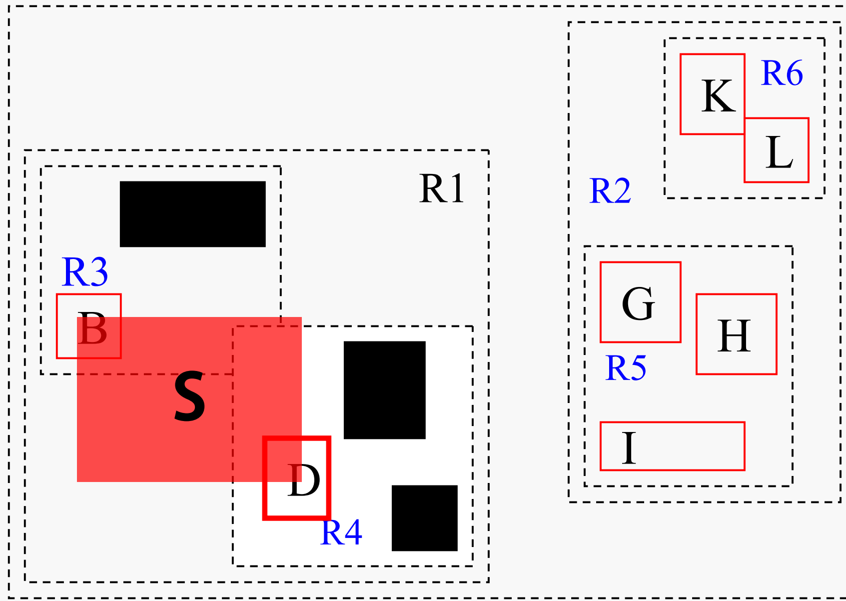


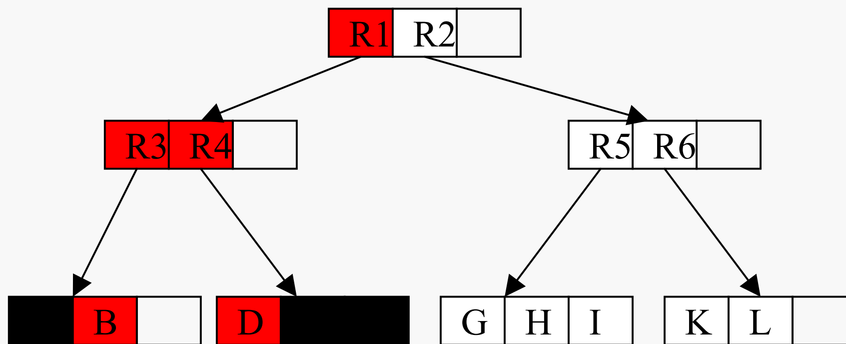
R-Tree Search Example (5/7)



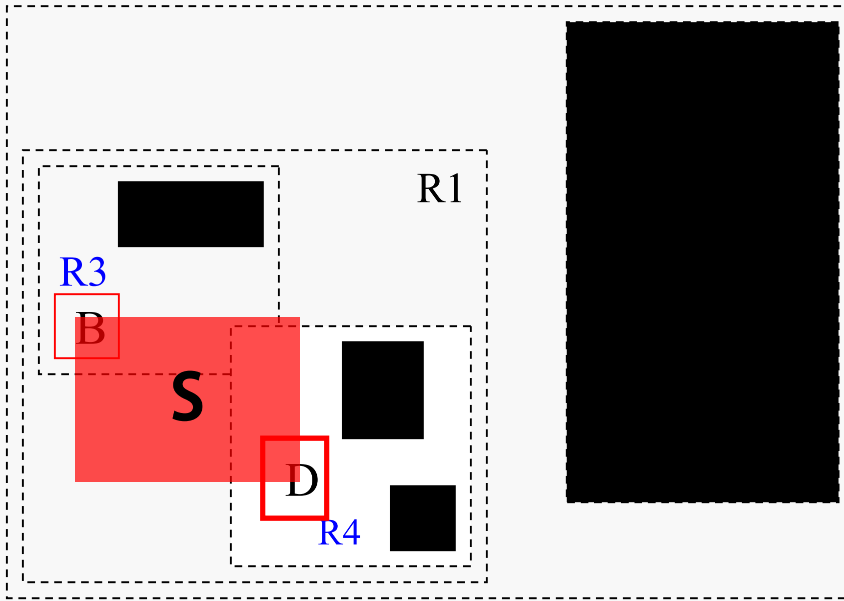


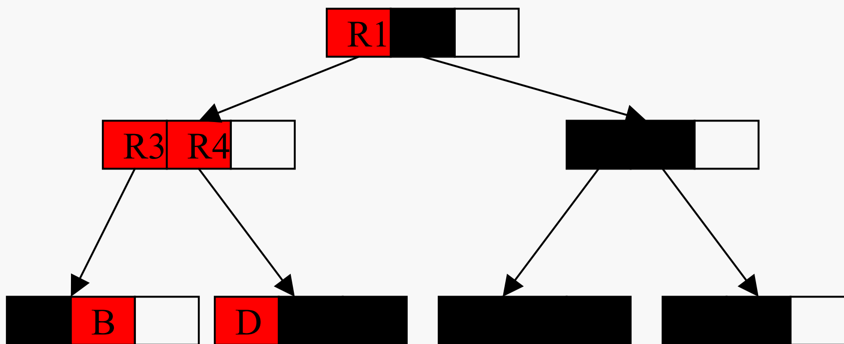
R-Tree Search Example (6/7)





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.

  - [Find position for new record]
    - Invoke ChooseLeaf to select a leaf node L in which to place E.
  - [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.
  - [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

// 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

// 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

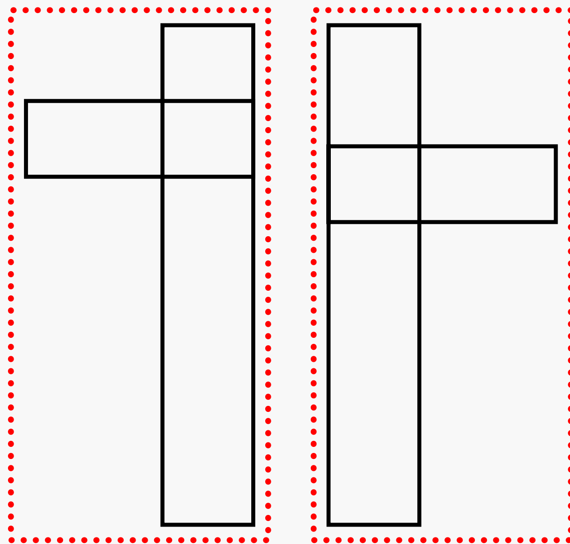
// Given a leaf node L from which an entry has been deleted, eliminate the node if it
// has too few entries and relocate its entries.
// Propagate node elimination upward as necessary.
// Adjust all covering rectangles on the path to the root.

- [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.

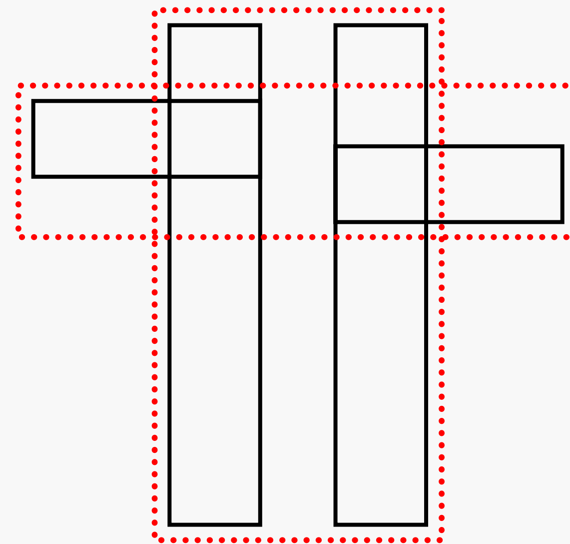
    Entries from eliminated leaf nodes are re-inserted in tree leaves as described in Insert, but entries from higher-level nodes must be placed higher in the tree, so that leaves of their dependent subtrees will be on the same level as leaves of the main tree.

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.



bad split



good split

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)

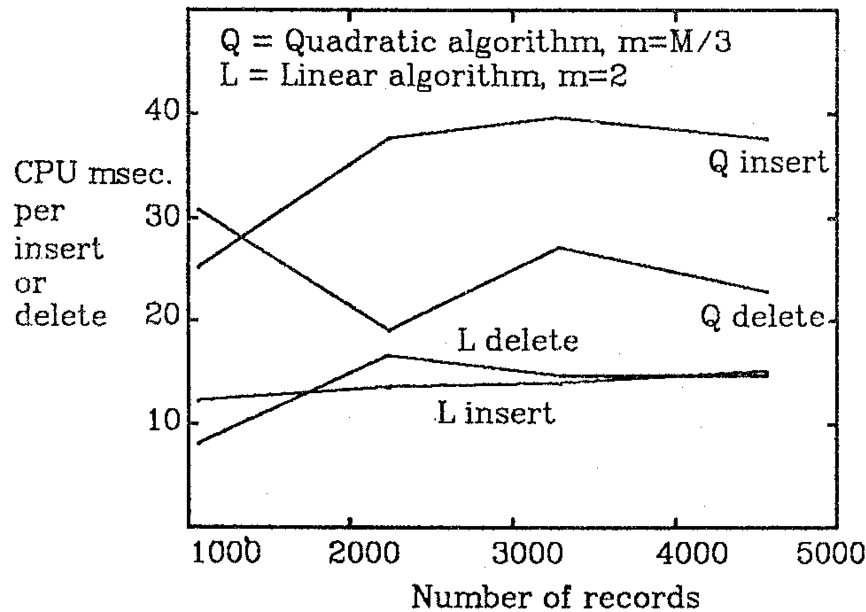


Figure 4.7
CPU cost of inserts and deletes vs. amount of data.

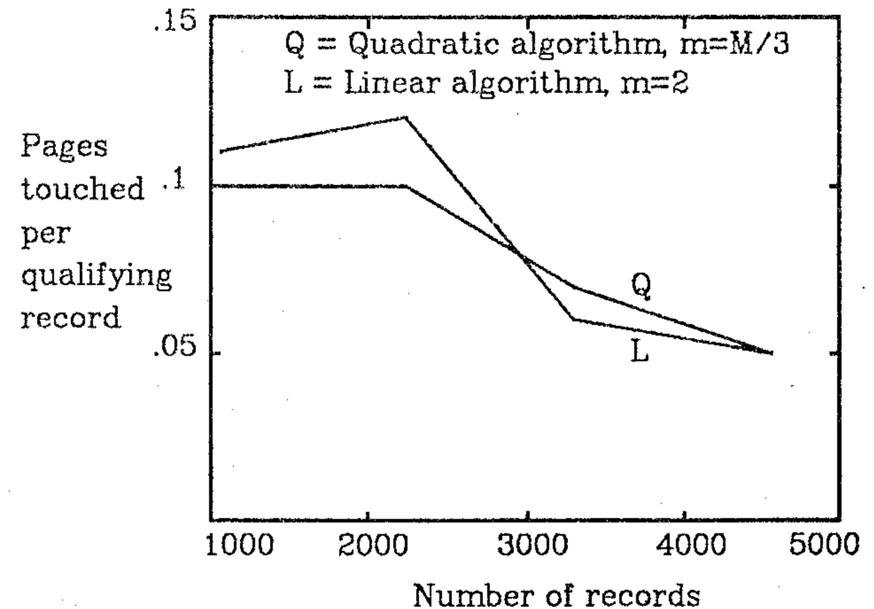


Figure 4.8
Search performance vs. amount of data: Pages touched

Performance (Search/Space)

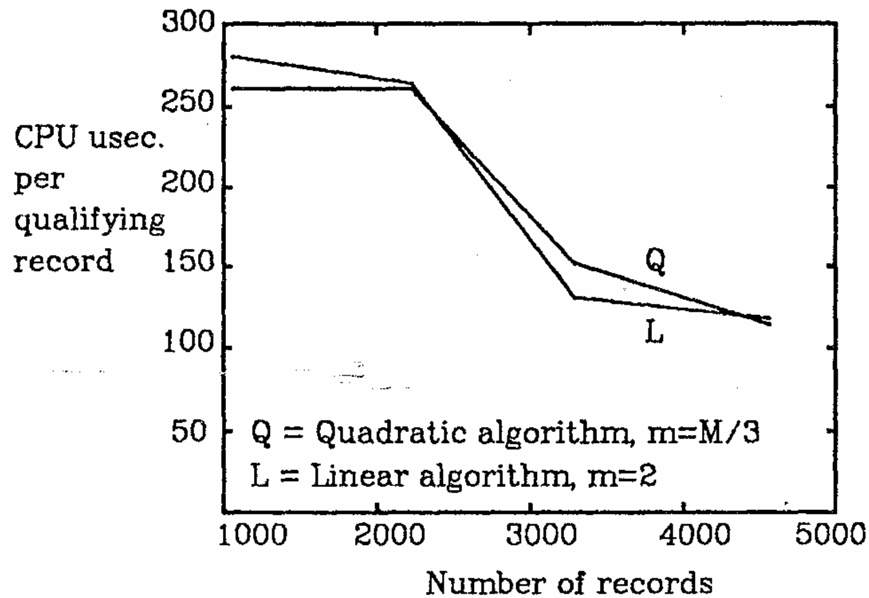


Figure 4.9
Search performance vs. amount of data:
CPU cost

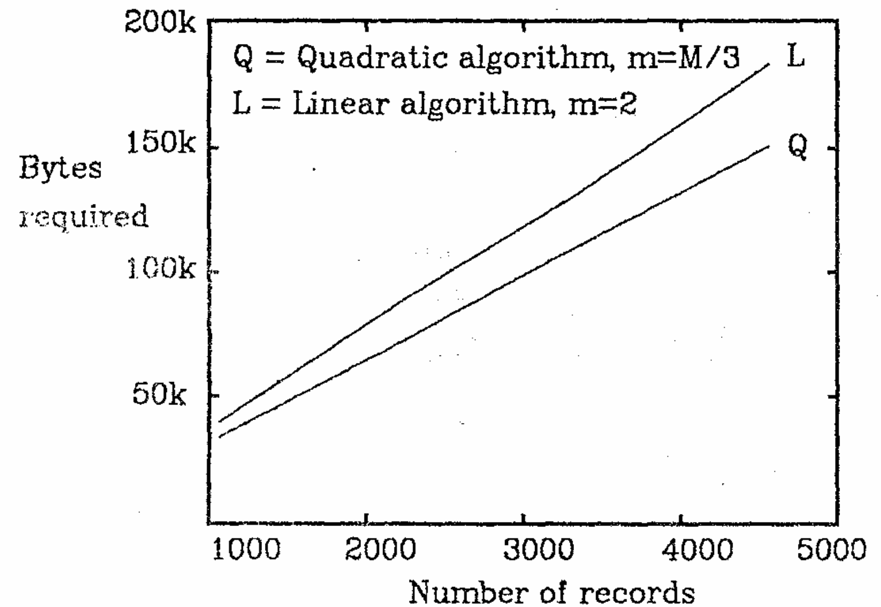


Figure 4.10
Space required for R-tree
vs. amount of data.

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.

N. Bechmann, H.-P. Kriegel, R. Schneider, B. Seeger:

“The R*-tree: An Efficient and Robust Access Methods for Points and Rectangles”,

ACM SIGMOD Conference, 1990.

Introduction

- R-tree
  - Tries to minimize the area of the enclosing rectangle (minimum bounding rectangle: MBR) in each inner node.

    [Note] SAMs are based on the approximation of a complex spatial object by MBR.
- R*-tree
  - Tries to minimize the area, margin and overlap of each enclosing rectangle in the directory and to increase storage utilization.

  ⇨ It clearly outperforms the R-tree.

Parameters for Retrieval Performance

- The area covered by a directory rectangle should be minimized.

  ⇨ The dead space should be minimized.

- The overlap between directory rectangles should be minimized.
- The margin of a directory rectangle should be minimized.
- Storage utilization should be maximized.

■ Overlap of an entry

$E_1, \ldots, E_p$ are the entries in a node.

$$\text{overlap}(E_k) = \sum_{i=1, i \neq k}^{p} \text{area}(\text{rectangle}(E_k) \cap \text{rectangle}(E_i)), i \leq k \leq p$$

Trade-offs in Performance Parameters

- Keeping the area, overlap, and margin of a directory rectangle small will be paid with lower storage utilization.
- Since more quadratic directory rectangles support packing better, it will be easier to maintain high storage utilization.
- The performance for queries with large query rectangles will be affected more by the storage utilization.

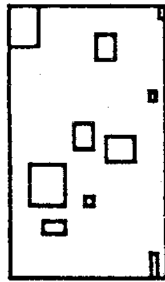


Figure 1a: Overfilled node

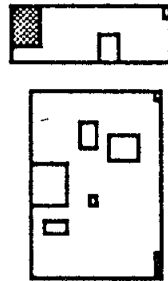


Figure 1b: Split of the quadratic R-tree, m = 30%

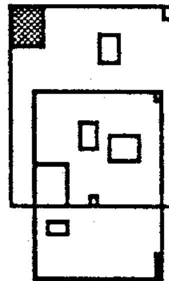


Figure 1c: Split of the quadratic R-tree, m = 40%

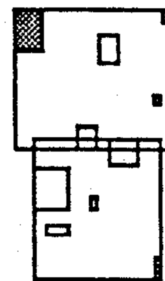


Figure 1d: Greene's split

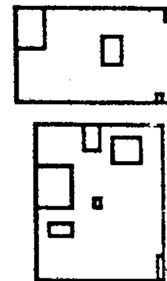


Figure 1e: Split of the R*-tree, m = 40%

Split of the R*-tree

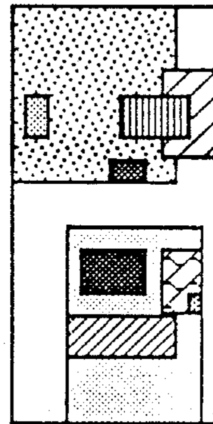


Figure 2a: Overfilled node

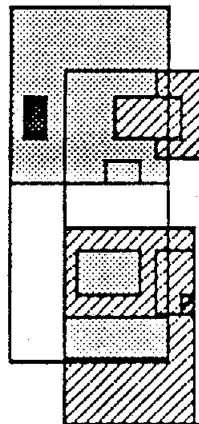


Figure 2b: Greene's split where the splitaxis is horizontal

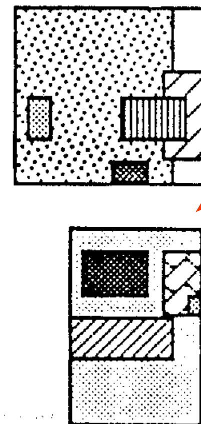


Figure 2c: Split of the R*-tree where the splitaxis is vertical

Algorithm ChooseSubtree

- Set N to be the root.
- If N is a leaf, return N. *
- Else
  - If the child_pointers in N point to leaves
    // Determine the minimum overlap cost //
    - Choose the entry in N whose rectangle needs least overlap enlargement to include the new data rectangle.
    - Resolve ties by choosing the entry whose rectangle needs least area enlargement, then the entry with the rectangle of smallest area.
  - Else
    // Determine the minimum area cost //
    - Choose the entry in N whose rectangle needs least area enlargement to include the new data rectangle.
    - Resolve ties by choosing the entry with the rectangle of smallest area.
- Set N to be the child_node pointed to by the child_pointer of the chosen entry and repeat from *.

Split of the R*-tree

- Finding good splits
  - Along each axis, the entries are first sorted by the lower value, then sorted by the upper value of their rectangles.
  - For each sort M-2m+2 distributions of the M+1 entries into 2 groups are determined, where the k-th distribution (k = 1, …, M-2m+2) is described as follows:
    - The 1st group contains the 1st (m-1)+k entries,
    - The 2nd group contains the remaining entries.

1, 2, 3, …, m, m+1, …, M-m+1, M-m+2, …, M+1

m entries | possible M-2m+2 splits | m entries

  - For each distribution, goodness values are determined.
    - Depending on these goodness values the final distribution of the entries is determined.
    - Three different goodness values and different approaches of using them in different combinations are tested experimentally.

Three Goodness Values

- Area-value
  - area[bb(1st group)] + area[bb(2nd group)]
- Margin-value
  - margin[bb(1st group)] + margin[bb(2nd group)]
- Overlap-value
  - area[bb(1st group)] $\cap$ area[bb(2nd group)]

where bb: bounding box of a set of rectangles

The Split Process

Algorithm Split

- Invoke ChooseSplitAxis to determine the axis, perpendicular to which the split is performed.
- Invoke ChooseSplitIndex to determine the best distribution into 2 groups along that axis.
- Distribute the entries into 2 groups.

Algorithm ChooseSplitAxis

- For each axis
  - Sort the entries by the lower then by the upper value of their rectangles and determine all distributions as described above.
  - Compute S, the sum of all margin-values of the different distributions.
- Choose the axis with the minimum S as split axis.

Algorithm ChooseSplitIndex

- Along the chosen split axis, choose the distribution with the minimum overlap-value.
- Resolve ties by choosing the distribution with minimum area-value.

Forced Reinsert

- Data rectangles inserted during the early growth of the index may have introduced directory rectangles not suitable to the current situation.
- This problem would be maintained or even worsened if underfilled nodes would be merged under the old parent.
- R-tree
  - Deletes the node and reinserts the orphaned entries in the corresponding level.
  - ⇨ Distributing entries into different nodes.
- R*-tree
  - Forces entries to be reinserted during the insertion routine.

Insertion

Algorithm Insert

- Invoke ChooseSubtree, with the level as a parameter, to find an appropriate node N, in which to place the new entry E.
- If N has less than M entries, accommodate E in N. If N has M entries, invoke OverflowTreatment with the level of N as a parameter [for reinsertion or split].
- If OverflowTreatment was called and a split was performed, propagate OverflowTreatment upwards if necessary. If OverflowTreatment caused a split of the root, create a new root.
- Adjust all covering rectangles in the insertion path s.t. they are minimum bounding boxes enclosing their children rectangles.

Algorithm OverflowTreatment

- If the level is not the root level and this is the 1st call of OverflowTreatment in the given level during the insertion of one data rectangle, then invoke ReInsert.
- Else Invoke Split.

Reinsert

Algorithm ReInsert

1. For all M+1 entries of a node N, compute the distance between the centers of their rectangles and the center of the bounding rectangle of N.
2. Sort the entries in decreasing order of their distances computed in 1.
3. Remove the first p entries from N and adjust the bounding rectangle of N.
4. In the sort, defined in 2, starting with the maximum distance (= far reinsert) or minimum distance (= close reinsert), invoke Insert to reinsert the entries.

- The parameter p can be varied independently as part of performance tuning.

  ⇨ Experiments showed that p = 30% of M yields the best performance.

Performance Comparison

Rectangle Data File	Query Average	Spatial Join	Storage Util.	Insertion
R-tree w/ linear split cost	227.5	261.2	62.7	12.63
R-tree w/ quad. split cost	130.0	147.3	68.1	7.76
Greene's R-tree	142.3	171.3	89.7	7.67
R*-tree	100.0	100.0	73.0	6.13

Point Data File	Query Average	Storage Util.	Insertion
R-tree w/ linear split cost	233.1	64.1	7.34
R-tree w/ quad. split cost	175.9	67.8	4.51
Greene's R-tree	237.8	69.0	5.20
2-level GRID	127.6	58.3	2.56
R*-tree	100.0	70.9	3.36

Conclusions

- Forced reinsert changes entries between neighboring nodes and thus decreases the overlap.
- As a side effect, storage utilization is improved.
- Due to more restructuring, less splits occur.
- Since the outer rectangles of a node are reinserted, the shape of the directory rectangles will be more quadratic. As discussed above, this is a desirable property.

S. Berchtold, D.A. Keim, and H.-P. Kriegel:
“The X-tree: An Index Structure for High-Dimensional Data”,
VLDB Conference, 1996.

Motivation

The R*-tree is not adequate for indexing high-dimensional datasets.

- Major problem of R-tree-based indexing methods
  - The overlap of the bounding boxes in the directory
    ⇨ which increases with growing dimension.
- X-tree
  - Uses a split algorithm that minimizes overlap.
  - Uses the concept of supernodes.
  - Outperforms the R*-tree and the TV-tree by up to 2 orders of magnitude.

Introduction

- Some observations in high-dimensional datasets
  - Real data in high-dimensional space are highly correlated and clustered.

    ⇨ The data occupy only some subspace.
  - In most high-dimensional datasets, a small number of dimensions bears most of the information.

  ⇨ Transforms data objects into some lower dimensional space

    ⇦ Traditional index structures may be used.
- Problems of Dimensionality Reduction
  - The datasets still have a quite large dimensionality.
  - It’s a static method.
- X-tree
  - Avoids overlap of bounding boxes in the directory by using a new organization of directory ⇨ supernode
  - Avoids splits which would result in a high degree of overlap in the directory.
  - Instead of allowing splits that introduce high overlaps, directory nodes are extended over the usual block size, resulting in supernodes.

Problems of R-tree-based Index Structures

- The performance of the R*-tree deteriorates rapidly when going to higher dimensions ⇦ The overlap in the directory is increasing very rapidly with growing dimensionality of data.

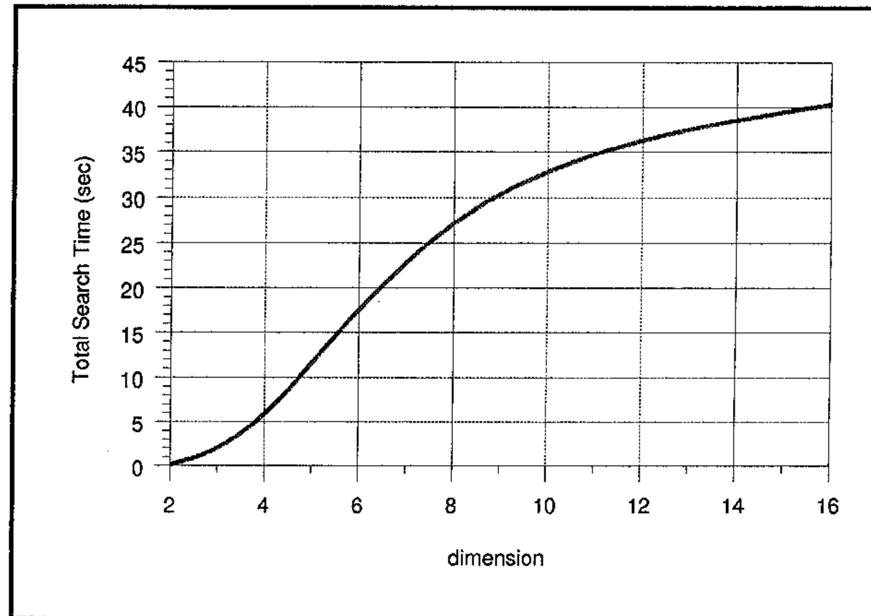
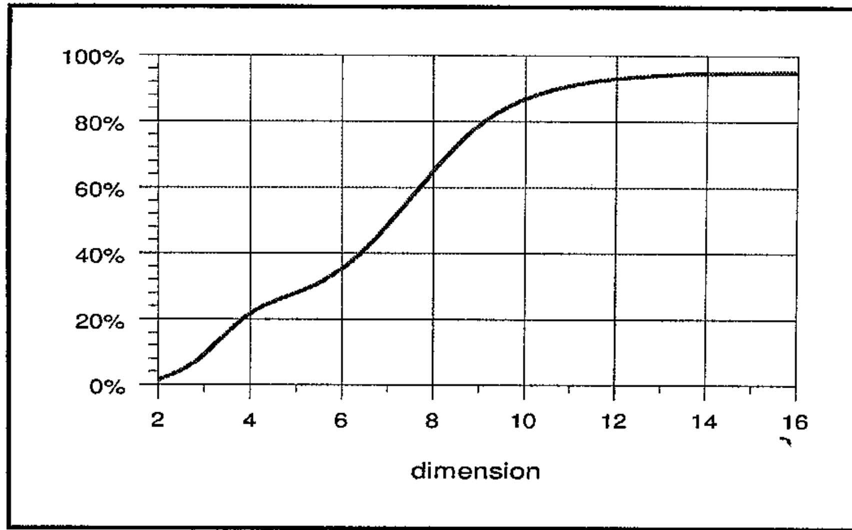
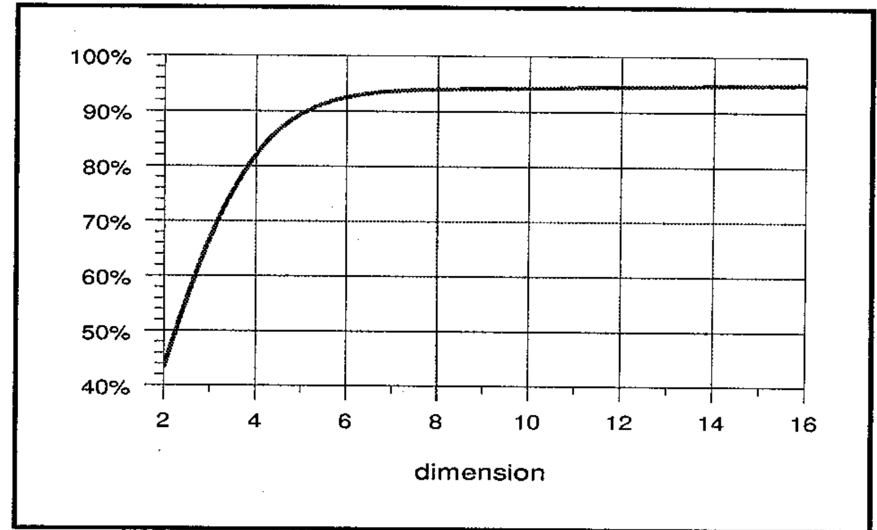


Figure 1: Performance of the R-tree Depending on the Dimension (Real Data)

Overlap of R*-tree Directory Nodes



a. Overlap (Uniformly Distributed Data)



b. Weighted Overlap (Real Data)

Figure 2: Overlap of R*-tree Directory Nodes depending on the Dimensionality

Definition of Overlap

- The overlap of an R-tree node is the percentage of space covered by more than one hyper-rectangle.

$$Overlap = \frac{\left\| \bigcup_{i,j \in \{1 \ldots n\}, i \neq j} (R_i \cap R_j) \right\|}{\left\| \bigcup_{i \in \{1 \ldots n\}} R_i \right\|}$$

- The weighted overlap of an R-tree node is the percentage of data objects that fall in the overlapping portion of the space.

$$WeightedOverlap = \frac{\left| \left\{ p \mid p \in \bigcup_{i,j \in \{1 \ldots n\}, i \neq j} (R_i \cap R_j) \right\} \right|}{\left| \left\{ p \mid p \in \bigcup_{i \in \{1 \ldots n\}} R_i \right\} \right|}$$

Multi-overlap of an R-tree Node

- The sum of overlapping volumes multiplied by the number of overlapping hyper-rectangles relative to the overall volume of the considered space.

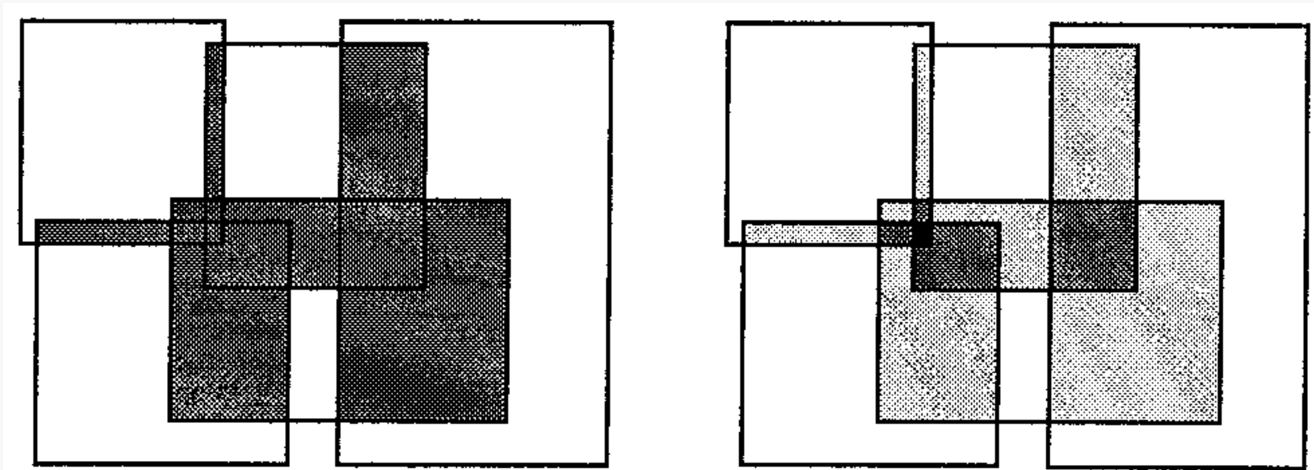


Figure 3: Overlap and Multi-Overlap of 2-dimensional data

X-tree (eXtended node tree)

- The X-tree avoids overlap whenever possible without allowing the tree to degenerate.
- Otherwise, the X-tree uses extended variable size directory nodes, called supernodes.
- The X-tree may be seen as a hybrid of a linear array-like and a hierarchical R-tree-like directory.
  - In low dimensions
    ⇨ a hierarchical organization would be most efficient.
  - For very high dimensionality
    ⇨ a linear organization is more efficient.
  - For medium dimensionality
    ⇨ Partially hierarchical and partially linear organization may be efficient.

Structure of the X-tree

- The X-tree consists of 3 kinds of nodes
  - Data nodes
  - Normal directory nodes
  - Supernodes: large directory nodes of variable size ⇨ To avoid splits in directory nodes

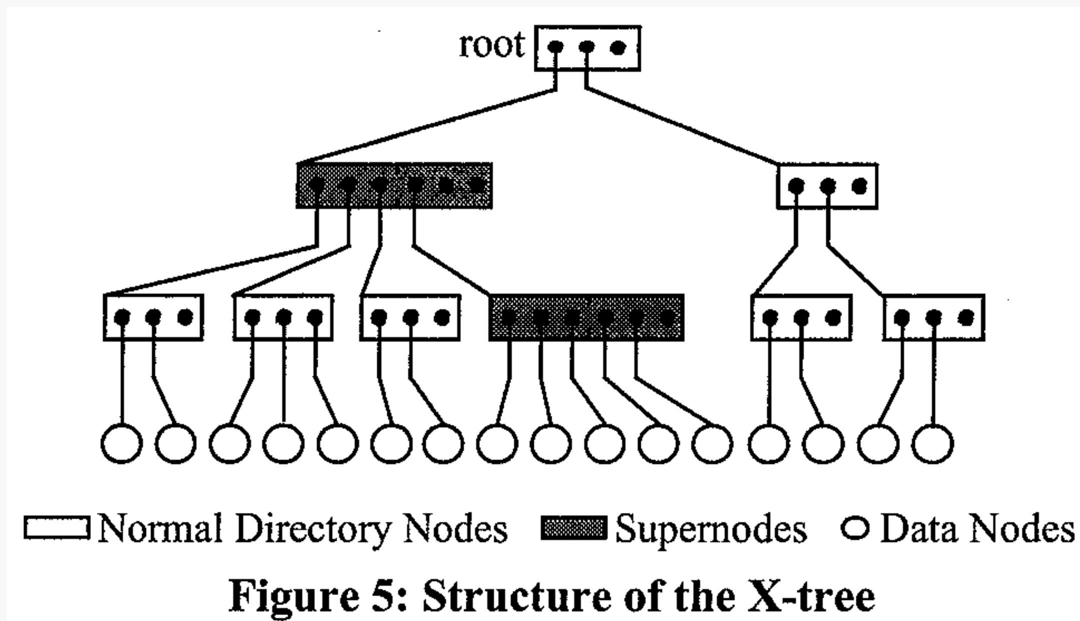


Figure 5: Structure of the X-tree

X-tree shapes in different dimensions

- The number and size of superndoes increases with the dimension.

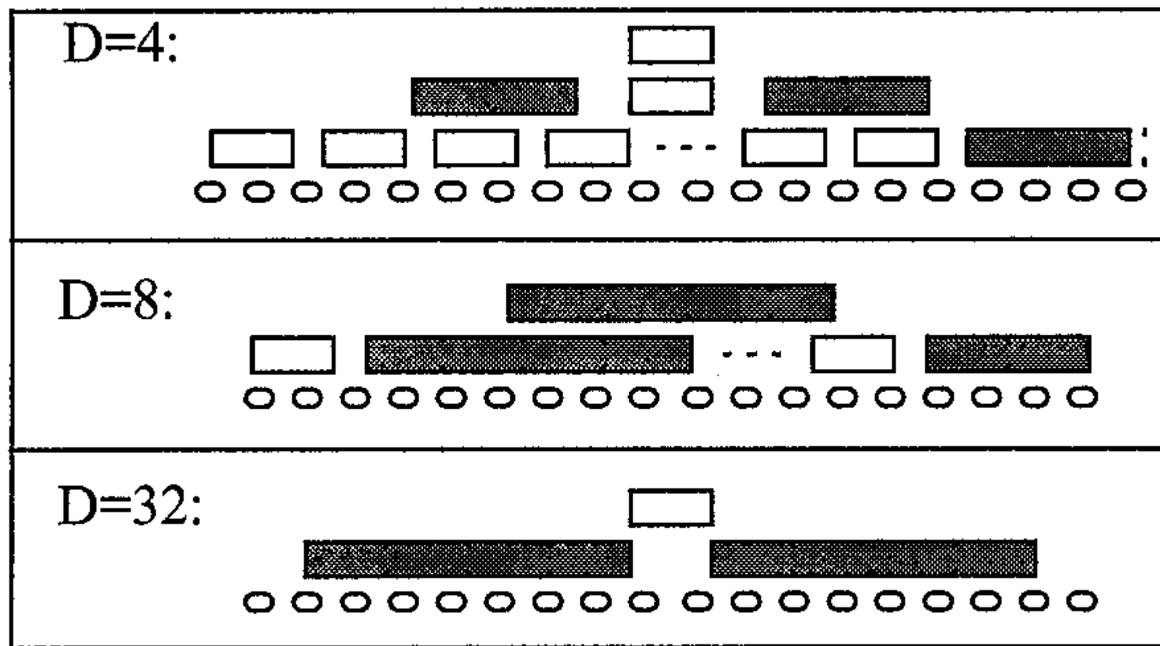


Figure 6: Various Shapes of the X-tree in different dimensions

X-tree Insertion

```
int X_DirectoryNode::insert(DataObject obj, X_Node **new_node)
{
    SET_OF_MBR *s1, *s2;
    X_Node *follow, *new_son;
    int return_value;

    follow = choose_subtree(obj);        // choose a son node to insert obj into
    return_value = follow->insert(obj, &new_son); // insert obj into subtree
    update_mbr(follow->calc_mbr());      // update MBR of old son node

    if (return_value == SPLIT){
        add_mbr(new_son->calc_mbr());    // insert mbr of new son node into current node

        if (num_of_mbrs() > CAPACITY){   // overflow occurs
            if (split(mbrs, s1, s2) == TRUE){
                // topological or overlap-minimal split was successfull
                set_mbrs(s1);
                *new_node = new X_DirectoryNode(s2);

                return SPLIT;
            }
            else // there is no good split
            {
                *new_node = new X_SuperNode();
                (*new_node)->set_mbrs(mbrs);

                return SUPERNODE;
            }
        }
    } else if (return_value == SUPERNODE){ // node 'follow' becomes a supernode
        remove_son(follow);
        insert_son(new_son);
    }

    return NO_SPLIT;
}
```

Split Algorithm

```
bool X_DirectoryNode::split(SET_OF_MBR *in, SET_OF_MBR *out1, SET_OF_MBR *out2)
{
    SET_OF_MBR t1, t2;
    MBR r1, r2;

    // first try topological split, resulting in two sets of MBRs t1 and t2
    topological_split(in, t1, t2);
    r1 = t1->calc_mbr(); r2 = t2->calc_mbr();

    // test for overlap
    if (overlap(r1, r2) > MAX_OVERLAP)
    {
        // topological split fails -> try overlap minimal split
        overlap_minimal_split(in, t1, t2);

        // test for unbalanced nodes
        if (t1->num_of_mbrs() < MIN_FANOUT || t2->num_of_mbrs() < MIN_FANOUT)
            // overlap-minimal split also fails (-> caller has to create supernode)
            return FALSE;
    }

    *out1 = t1; *out2 = t2;
    return TRUE;
}
```

Supernodes

- If the number of MBRs in one of the partitions is below a given threshold, the split algorithm terminates without providing a split.
- In this case, the current node is extended to become a supernode of twice the standard block size.
- If the same case occurs for an already existing supernode, the supernode is extended by one additional block.
- If a supernode is created or extended, there may be not enough contiguous space on disk to sequentially store the supernode. In this case, the disk manager has to perform a local reorganization.

Determining the Overlap-Minimal (Overlap-Free) Split

Definition 2 (Split)

The split of a node $S = \{mbr_1, \ldots, mbr_n\}$ into two subnodes $S_1 = \{mbr_{i_1}, \ldots, mbr_{i_{s_1}}\}$ and $S_2 = \{mbr_{i_1}, \ldots, mbr_{i_{s_2}}\}$ ($S_1 \neq \varnothing$ and $S_2 \neq \varnothing$) is defined as

$$Split(S) = \{(S_1, S_2) \mid S = S_1 \cup S_2 \wedge S_1 \cap S_2 = \varnothing\}.$$

The split is called

(1) overlap-minimal iff $\|MBR(S_1) \cap MBR(S_2)\|$ *is minimal*

(2) overlap-free iff $\|MBR(S_1) \cap MBR(S_2)\| = 0$

(3) balanced iff $-\varepsilon \leq |S_1| - |S_2| \leq \varepsilon$.

Lemma 1

For uniformly distributed point data, an overlap-free split is only possible iff there is a dimension according to which all MBRs in the node have been previously split. More formally,

Split(S) is overlap-free $\Leftrightarrow$
$\exists\, d \in \{1, \ldots, D\}\ \forall mbr \in S$:
mbr has been split according to d

Split History

- For finding an overlap-free split, we have to determine a dimension according to which all MBRs of *S* have been split previously.
  - ⇨ The split history provides the necessary information:
    - split dimensions and new MBRs created by split.
    - It may be represented by a binary tree, called the split tree.

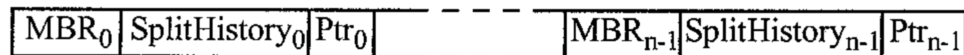


Figure 4: Structure of a Directory Node

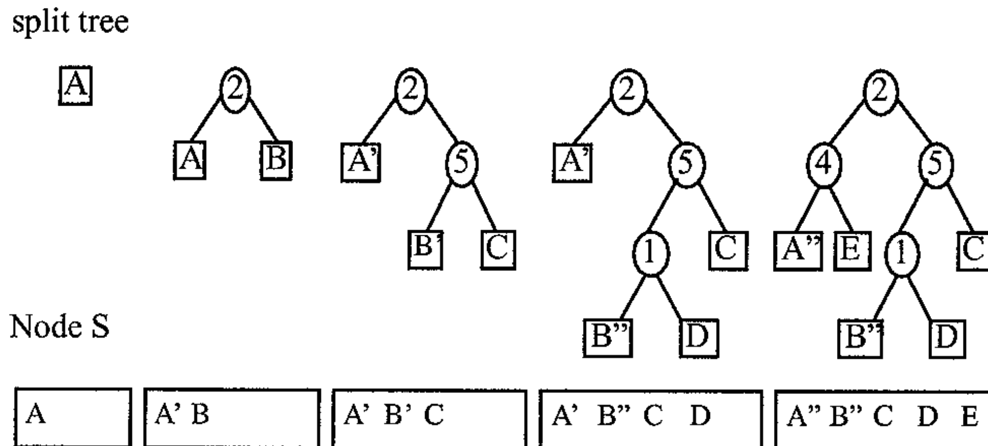
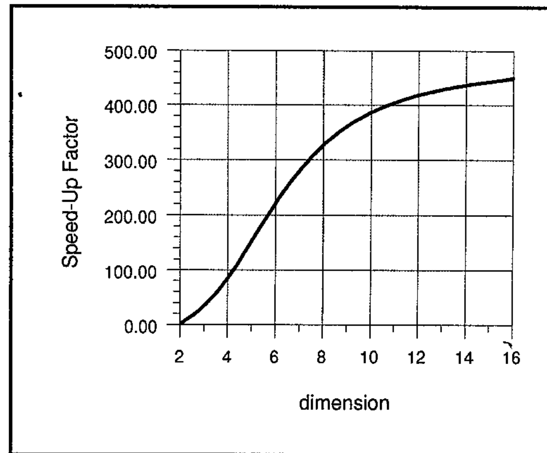
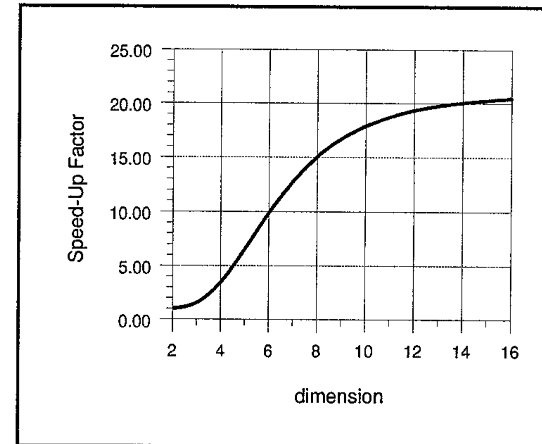


Figure 9: Example for the Split History

Performance Evaluation (1/2)



a. Point Query



b. 10 Nearest-Neighbor Query

Figure 11: Speed-Up of X-tree over R*-tree on Real Point Data (70 MBytes)

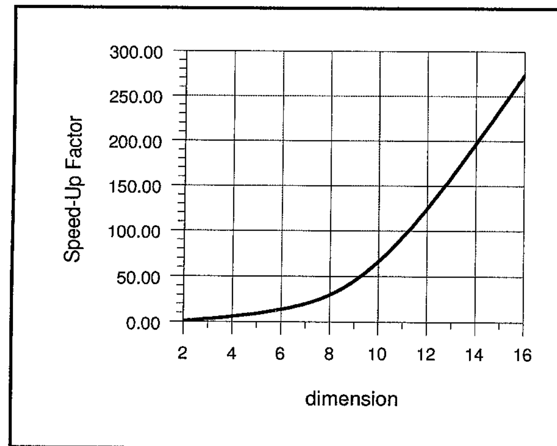


Figure 13: Speed-Up of X-tree over R*-tree on Point Queries (100 MBytes of Synthetic Point Data)

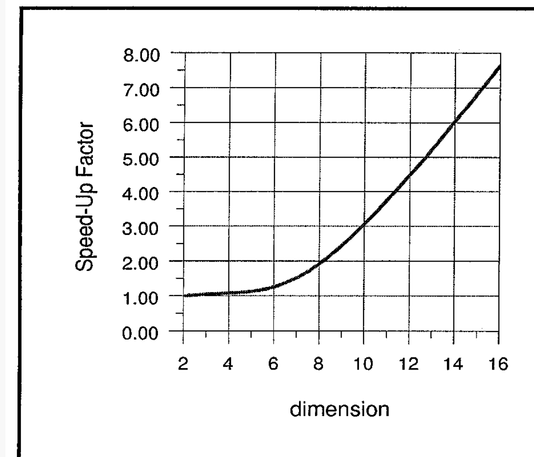
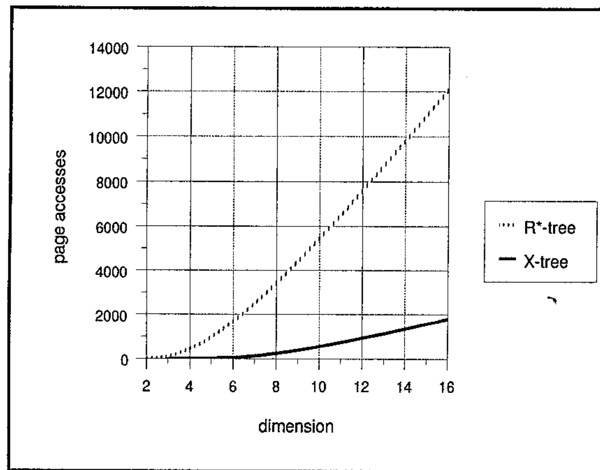
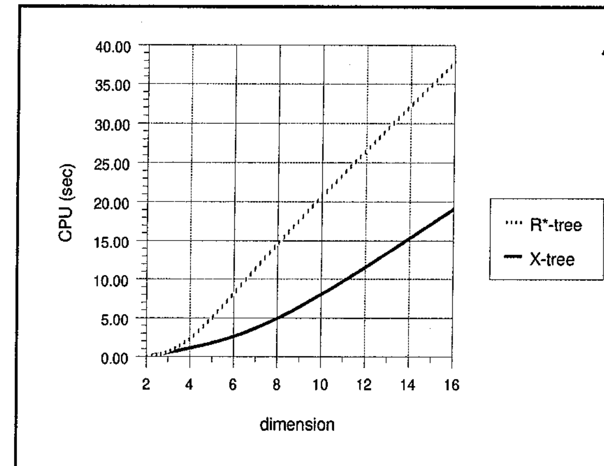


Figure 15: Speed-Up of X-tree over R*-tree on Real Extended Spatial Data

Performance Evaluation (2/2)



a. Page Accesses



b. CPU-Time

Figure 12: Number of Page Accesses versus CPU-Time on Real Point Data (70 MBytes)

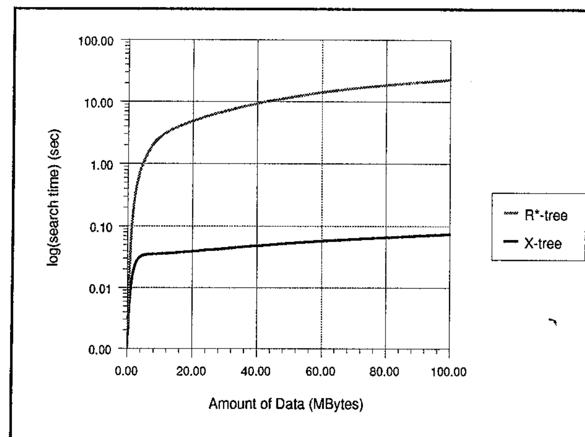


Figure 14: Total Search Time of Point Queries for Varying Database Size (Synthetic Point Data)

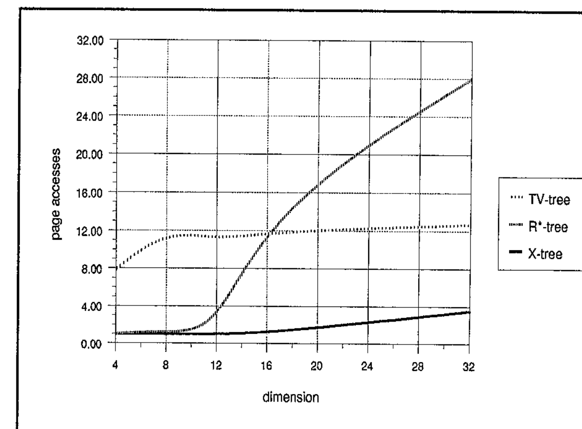


Figure 16: Comparison of X-tree, TV-tree, and R*-tree on Synthetic Data

S.Berchtold, C. Bohm, H.-P. Kriegel

"The Pyramid-Technique: Towards Breaking the Curse of Dimensionality"

ACM SIGMOD Conference, 1998.

Contents

- Introduction
- Analysis of Balanced Splits
- The Pyramid-Technique
- Query Processing
- Analysis of the Pyramid-Technique
- The Extended Pyramid-Technique
- Experimental Evaluation

Introduction

- A variety of new DB applications has been developed
  - Data warehousing
    - Require a multidimensional view on the data
  - Multimedia
    - Using some kind of feature vectors

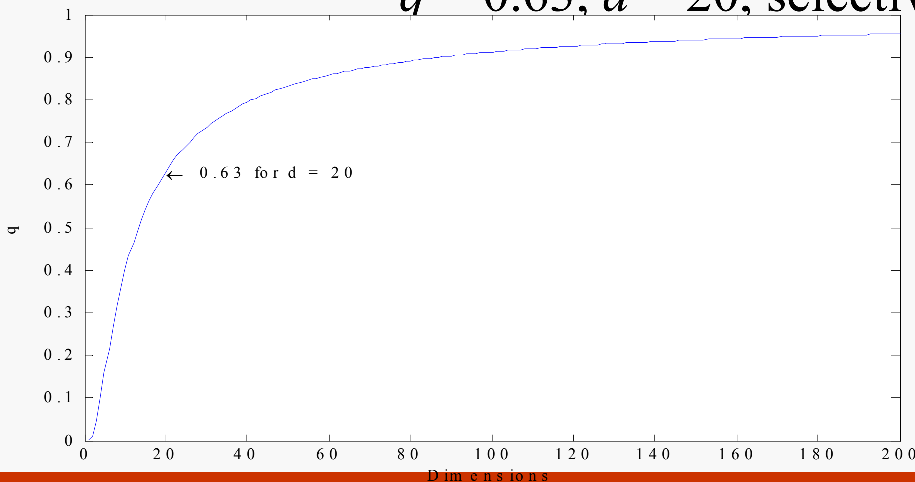
→ Has to support query processing on large amounts of high-dimensional data

Analysis of Balanced Splits

- Performance degeneration
  - Data space cannot be split in each dimension
    - To a uniformly distributed data set
    - 1-dimenstion data space → $2^1 = 2$ data pages
    - 20-dimension data space → $2^{20} \approx 1{,}000{,}000$ data pages

$$d' = \log_2\left(\frac{N}{C_{eff}}\right)$$

  - Similar property holds for range query
    - $q = \sqrt[d]{s}$
      - $q = 0.63$, $d = 20$, selectivity s = 0.01% = 0.0001



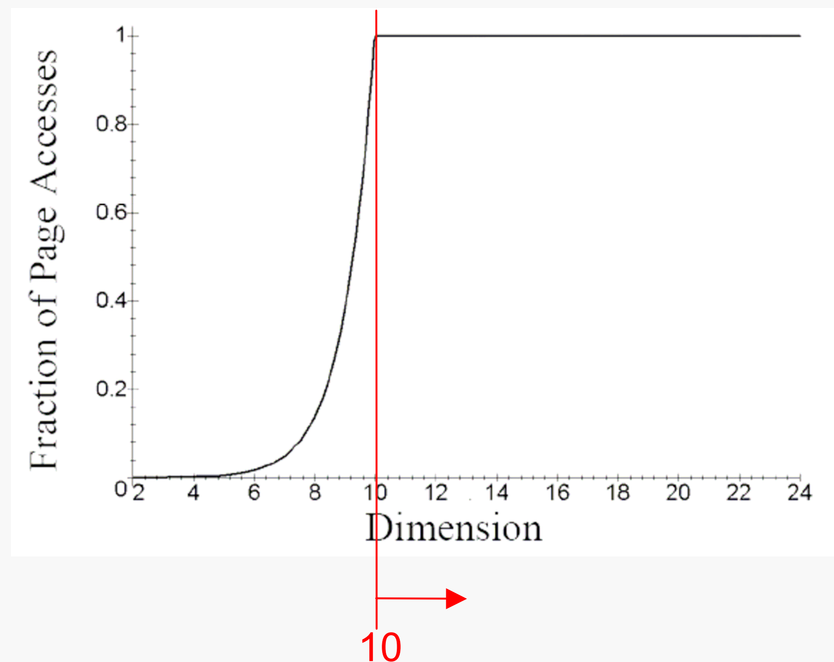
d=1: q = 0.0001
d=2: q = 0.01
d=3: q = 0.0464
…
d=20: q = 0.6310

Analysis of Balanced Splits

- Expected value of data page access

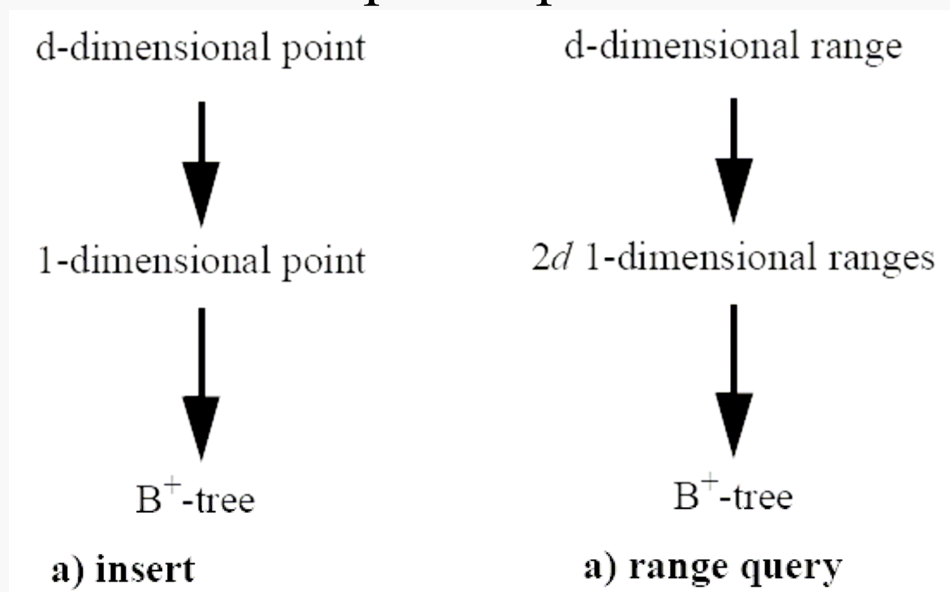
  –

$$E_{\text{balanced}(d,q,N)} = \frac{N}{C_{eff}(d)} \cdot \min\left(1, \left(\frac{0.5}{1-q}\right)^{\log_2\left(\frac{N}{C_{eff}(d)}\right)}\right)$$



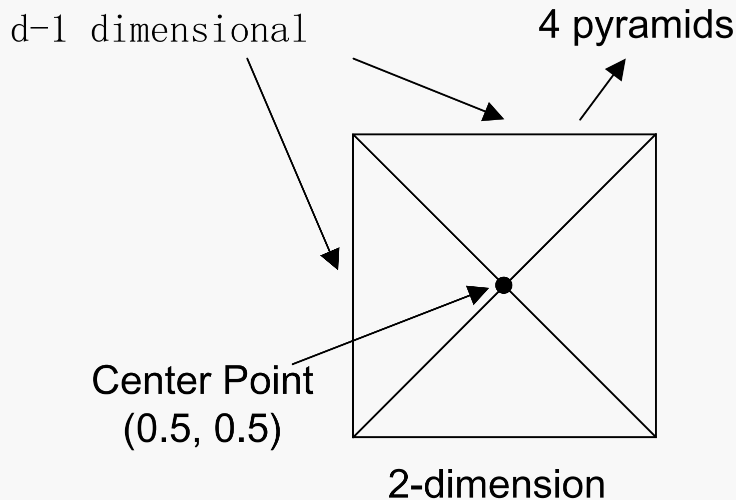
The Pyramid-Technique

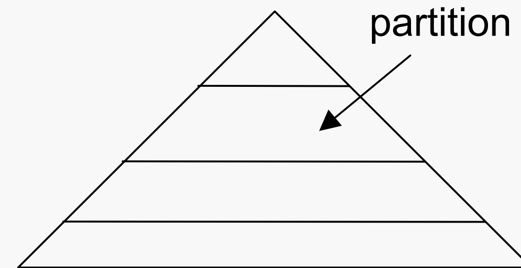
- Basic idea
  - Transform *d*-dimesional point into 1-dimensional value
  - Store and access the values using B^+-tree
    - Store d-dimensional points plus 1-dimensional key



Data Space Partitioning

- Pyramid-Technique
  - Split the data space into 2*d* pyramids
    - top: center point (0.5, 0.5, …, 0.5)
    - base: (d-1)-dimensional surface
  - each of pyramids is divided into several partitions





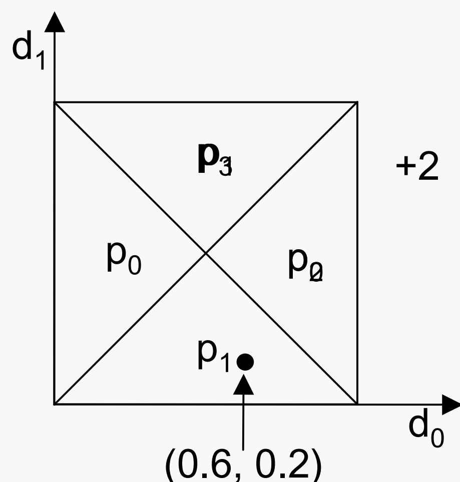
Data Space Partitioning

➢ Definition 1: (Pyramid of a point *v*)

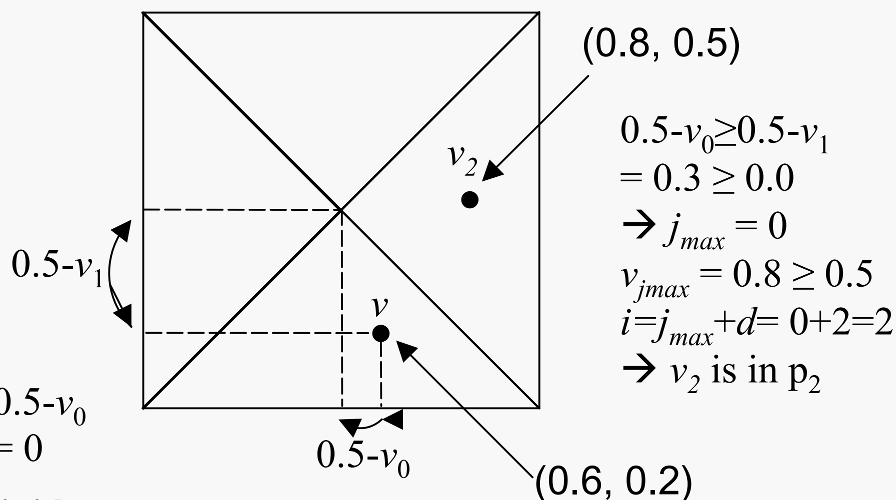
–

$$i = \begin{cases} j_{max} & if\,(v_{j_{max}} < 0.5) \\ (j_{max} + d) & if\,(v_{j_{max}} \geq 0.5) \end{cases}$$

- $$j_{max} = (j \,|\, (\forall k, 0 \leq (j,k) < d, j \neq k : |0.5 - v_j| \geq |0.5 - v_k|))$$



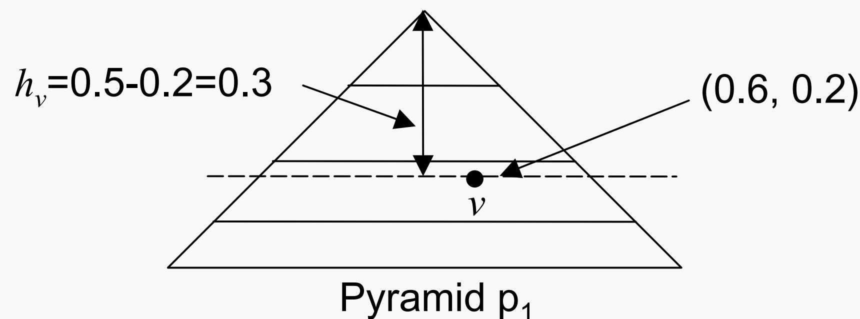
$0.5\text{-}v_1 \geq 0.5\text{-}v_0$
→ $j_{max} = 0$
$v_{jmax} = 0.2 < 0.5$
$i = 1$
→ v is in p_1



Data Space Partitioning

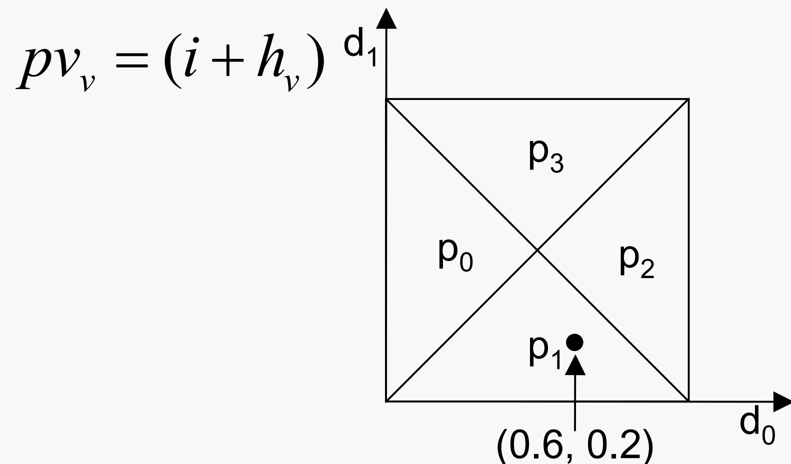
- Definition 2: (Height of a point *v*)
  –

$$h_v = \left|0.5 - v_{i \,\mathrm{MOD}\, d}\right|$$



- Definition 3: (Pyramid value of a point *v*)
  –

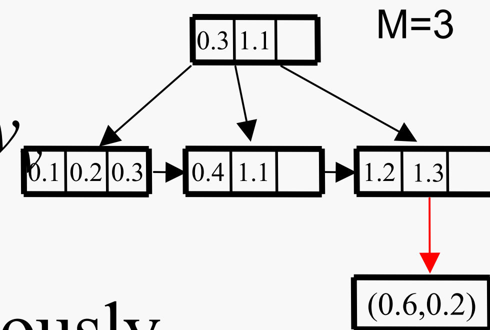
$$pv_v = (i + h_v)$$



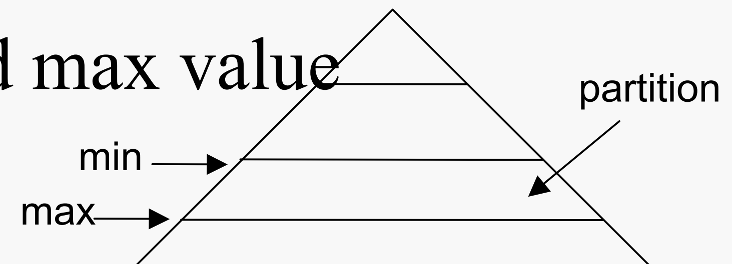
Def 1: $i = 1$
Def 2: $h_v = 0.3$
Def 3: $pv_v = 1{+}0.3 = 1.3$

Index Creation

- Insert
  - Determine the pyramid value of a point *v*
  - Insert into B$^+$-tree using pv_v as a key
  - Store the *d*-dimensional point *v* and pv_v
    - In the according data page of the B$^+$-tree
  - Update and delete can be done analogously

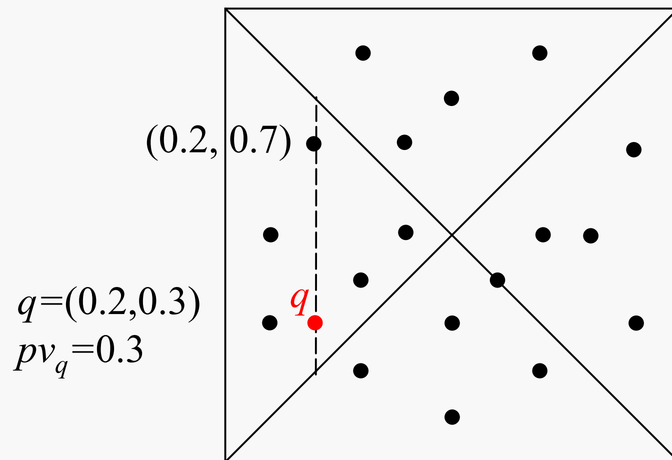


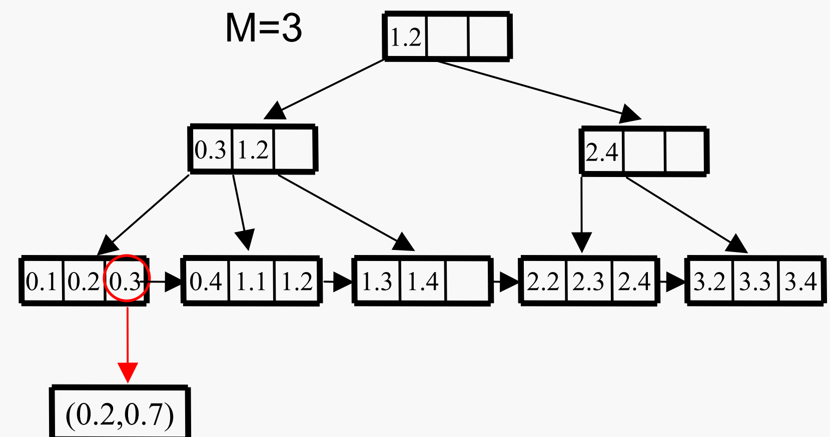
- Resulting data pages of the B$^+$-tree
  - Belong to same pyramid
  - Interval given by the min and max value



Query Processing

- Point query
  - Compute the pyramid value pv_q of q
  - Query the B^+-tree using pv_q
  - Obtain a set of points sharing pv_q
  - Determine whether the set contains q





Query Processing

➢ Range query

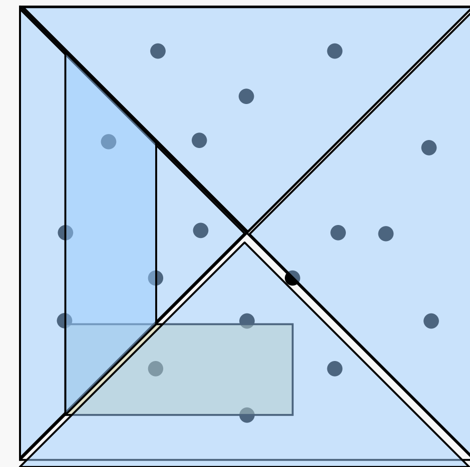
```
Point_Set PyrTree::range_query(range q)
{
  Point_Set res;
  for (i=0; i<2d; i++) {
    if (intersect(p[i], q) {
      determine_range(p[i], q, h_low, h_high);

      cs = btree_query(i+h_low, i+h_high);
      for (c=cs.first; cs.end; cs.next)
        if (inside(q, c))
          res.add(c);
    }
  }
  return res;
}
```

Lemma 1

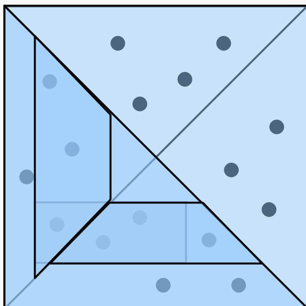
Lemma 2

i=0

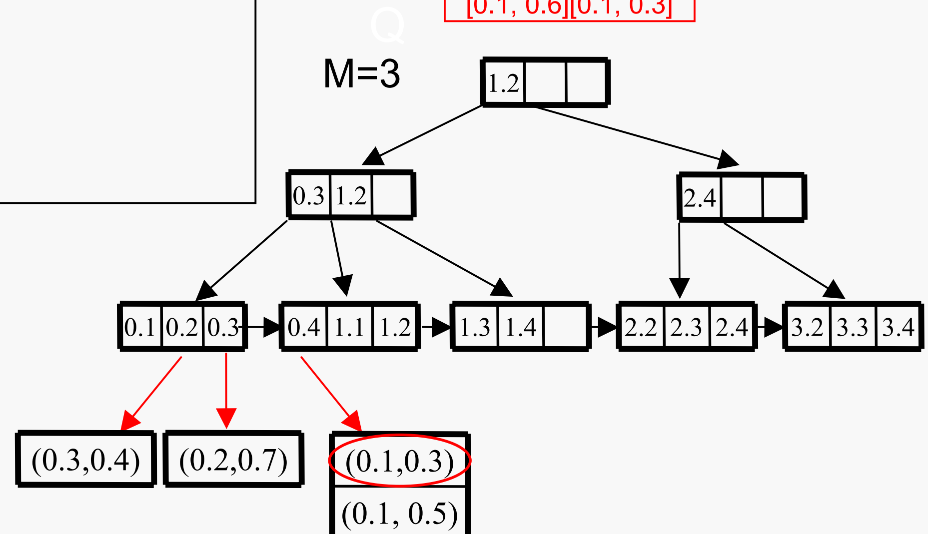
h_{low}=0.2
h_{high}=0.4



[0.1, 0.6][0.1, 0.3]



(0.1, 0.3)
(0.5, 0.3)
(0.3, 0.2)
(0.5, 0.1)

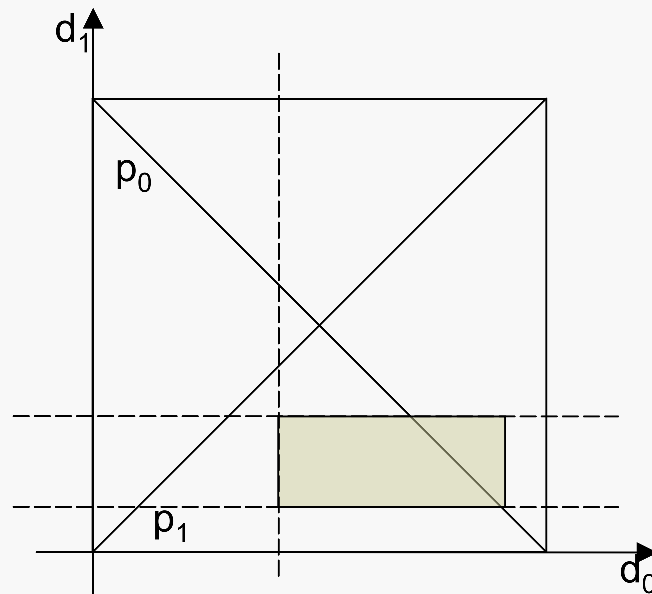


Query Processing

- Lemma 1: (Intersection of a Pyramid and a Rectangle)
  - $\forall j, 0 \le j < d, j \ne i: \ \hat{q}_{i_{min}} \le -MIN(\hat{q}_j)$

Only use i<d

$$MIN(r) = \begin{cases} 0 & \text{if } r_{min} \le 0 \le r_{max} \\ min(|r_{min}|, |r_{max}|) & \text{otherwise} \end{cases}$$



[0.4, 0.9][0.1, 0.3]

[-0.1, 0.4][-0.4, -0.2]

i=0:

q_{imin}=-0.1
-MIN(q_j)=-0.2
-0.1 > -0.2
→ false

i=1:

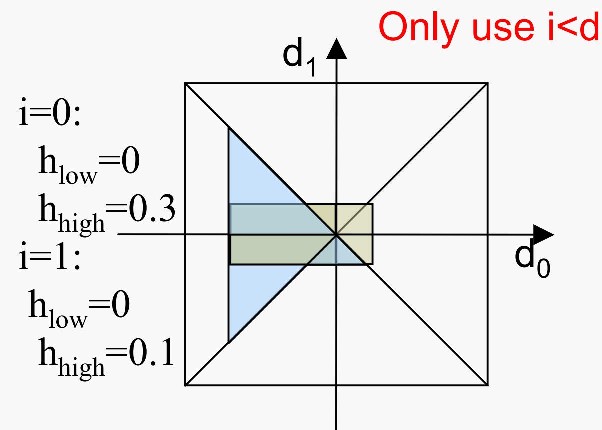
q_{imin}=-0.4
-MIN(q_j)=0
-0.4 < 0
→ true

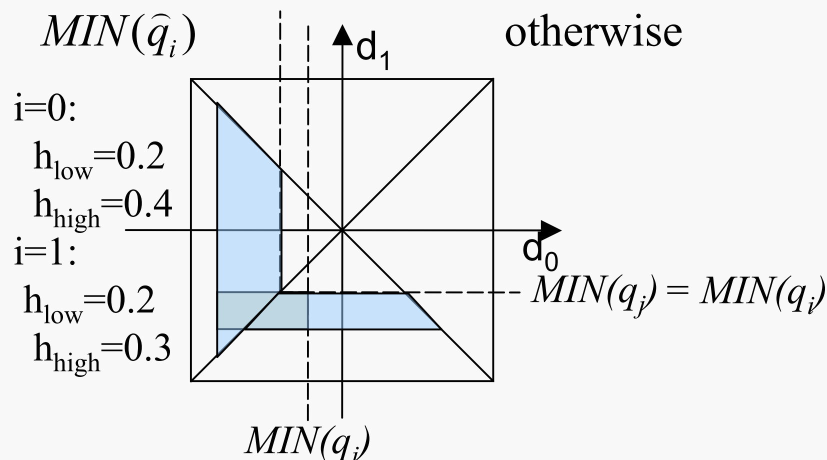
Query Processing

- Lemma 2: (Interval of Intersection of Query and Pyramid)
  - Case 1: $(\forall j, 0 \le j < d : (\hat{q}_{j_{min}} \le 0 \le \hat{q}_{j_{max}}))$
    - $h_{low} = 0,\ h_{high} = MAX(\hat{q}_i)$
  - Case 2: (otherwise)
    - $h_{low} = min_{(0 \le j < d, j \ne i)}(\bar{q}_{j_{min}}),\ h_{high} = MAX(\hat{q}_i)$

$$\bar{q}_{j_{min}} = \begin{cases} max(MIN(\hat{q}_i), MIN(\hat{q}_j)) & \text{if } MAX(\hat{q}_j) \ge MIN(\hat{q}_i) \\ MIN(\hat{q}_i) & \text{otherwise} \end{cases}$$

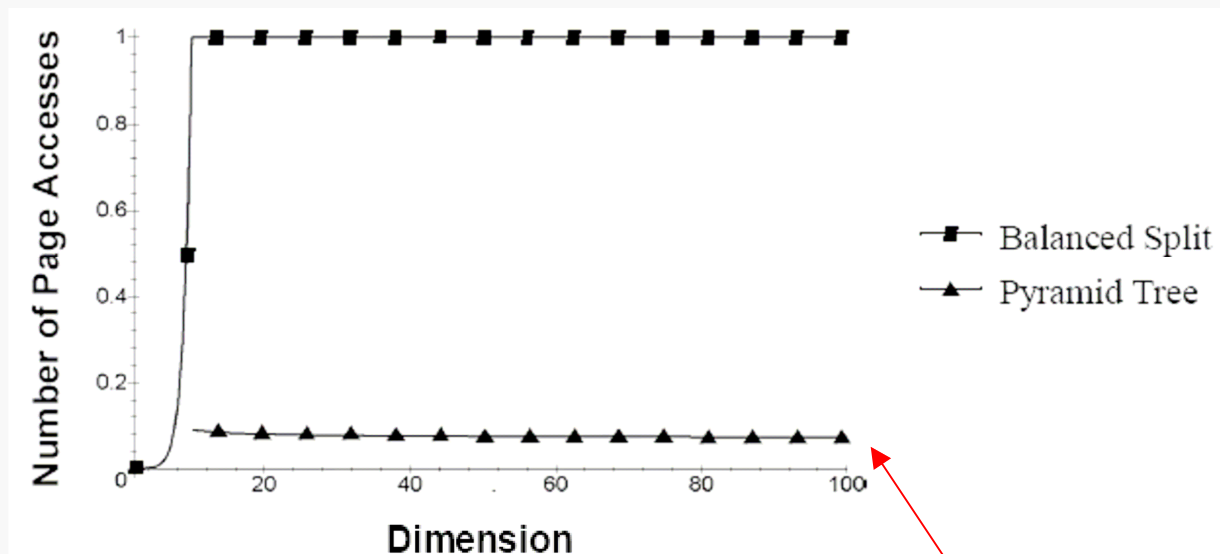
–





Analysis of the Pyramid-Technique

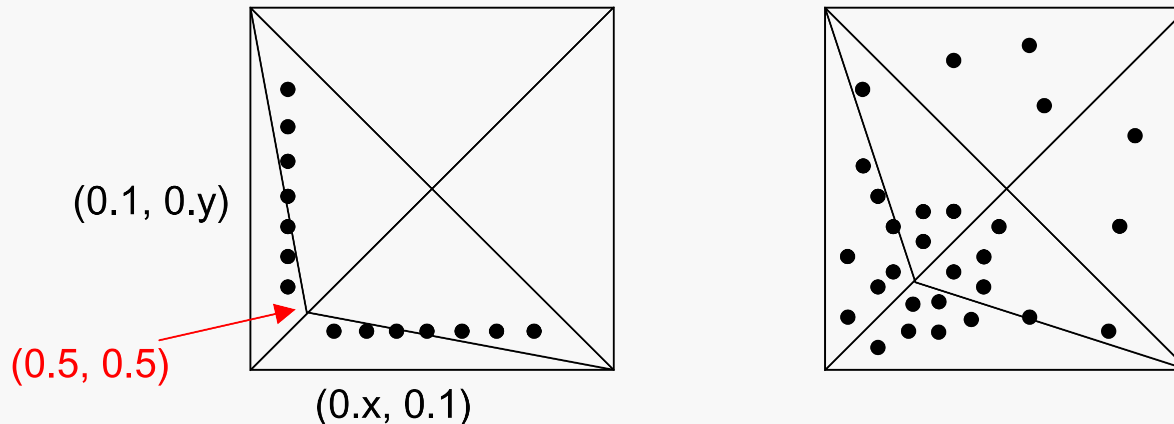
- Required number of accesses pages for 2*d* pyramids
  - $$E_{\text{pyrimidtree}}(d,q,N) = \frac{2d + N\cdot(1-(2q-1)^{d+1})}{2C_{eff}(d)\cdot(d+1)\cdot(1-q)}$$



Does not reveal any performance degeneration

The Extended Pyramid-Technique

- Basic idea

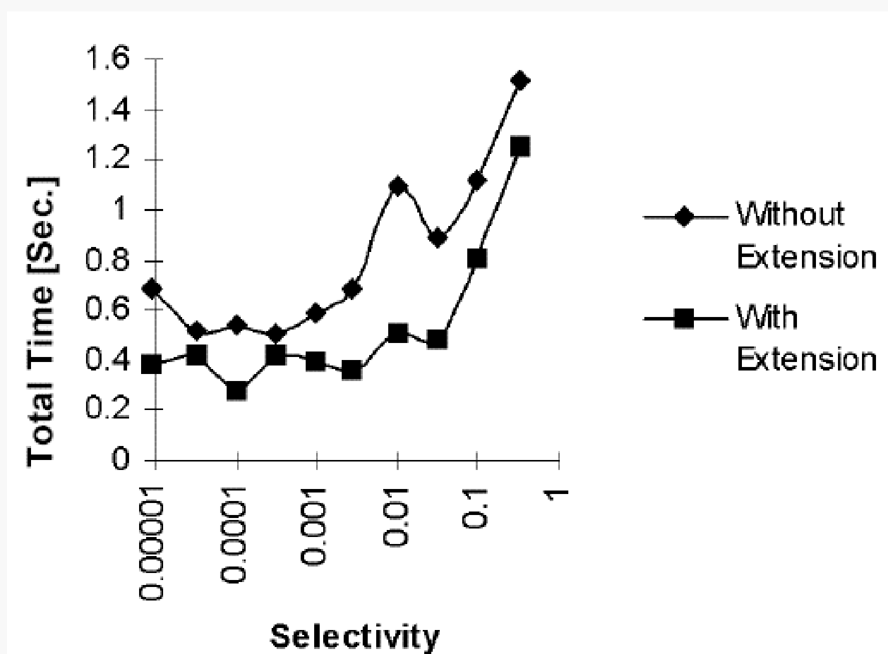


- Conditions
  -
  - $t_i(0)=0,\ t_i(1)=1,\ t_i(mp_i)=0.5,\ t_i:[0,1]\rightarrow[0,1]$
    - $t_i(x)=x^{-\frac{1}{\log_2(mp_i)}}$

      $t_i(x)=x^r,\quad t_i(mp_i)=0.5=mp_i^{\,r}$

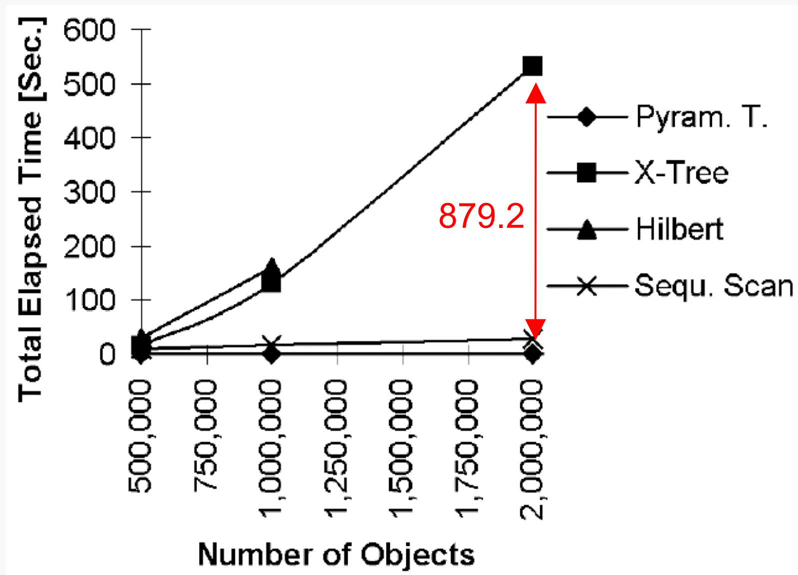
The Extended Pyramid-Technique

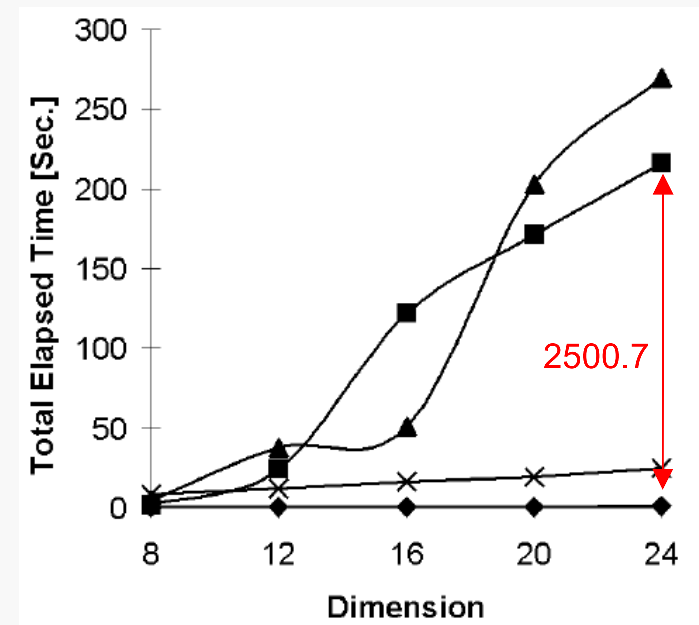
- Performance
  - Speed up : about 10 ~ 40%
  - Loss of performance not too high
    - Compare to high speed-up factors over other index structures



Experimental Evaluation

- Using Synthetic Data
  - Performance behavior over database size (16-dimension)

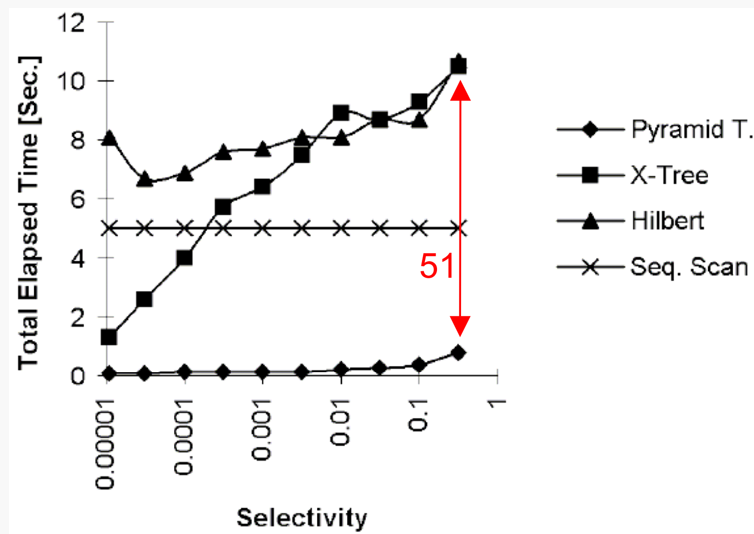




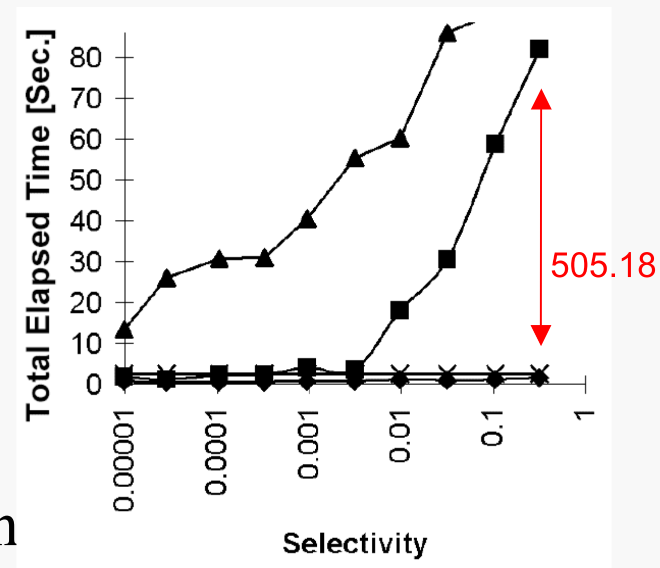
  - Performance behavior over data space dimension
    - 1,000,000 objects

Experimental Evaluation

- Using Real Data Sets
  - Query processing on text data (16-dimension)



  - Query processing on warehousin



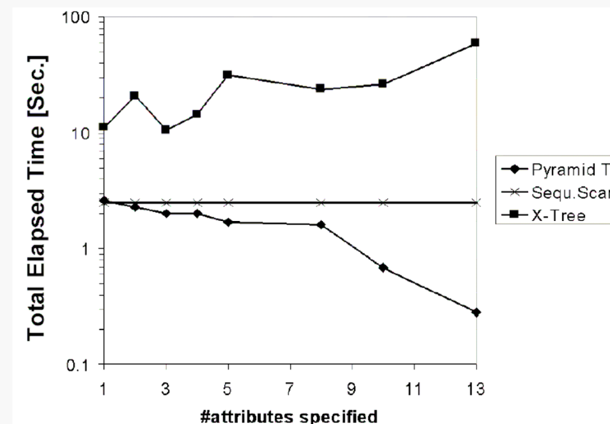
Conclusions

- For almost hypercube shaped queries
  - Outperforms any competitive technique
    - Including linear scan
  - Holds even for skewed, clustered and categorical data
- For queries having a bad selectivity
  - Outperforms competitive index structures
  - However, a linear scan is faster

<For 1-dimensional queries>

pyramid T.: 2.6

Sequ. Scan: 2.48



Fin.

N. Roussopoulos, S. Kelly, and F. Vincent:
“Nearest Neighbor Queries”,
ACM SIGMOD Conference, 1995.

Introduction

A frequently encountered type of query in MMDB and GIS is to find the *k* nearest neighbor objects (k NNs) to a given point in space.

Efficient branch-and-bound R-tree traversal algorithm to find *k* NNs to a point is presented.

Introduction

- Example k-NN queries
  - "Find k images most similar to a query image"
  - "Find k nearest hotels from this point"
- Efficient processing of k-NN queries requires spatial data structures which capitalize on the proximity of the objects to focus the search of potential neighbors only.
- An efficient branch-and-bound search algorithm for processing exact k-NN queries for the R-trees.

k-NN Search Using R-trees

- R-tree example
  - Minimum bounding rectangle (MBR)

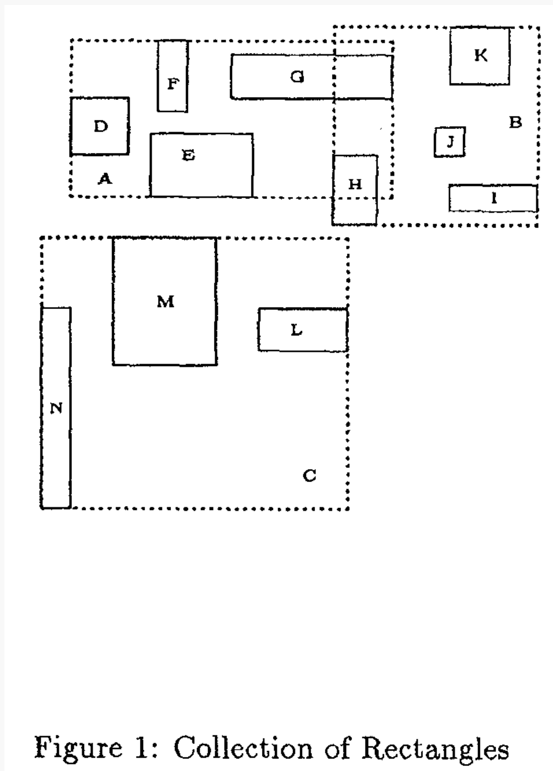


Figure 1: Collection of Rectangles

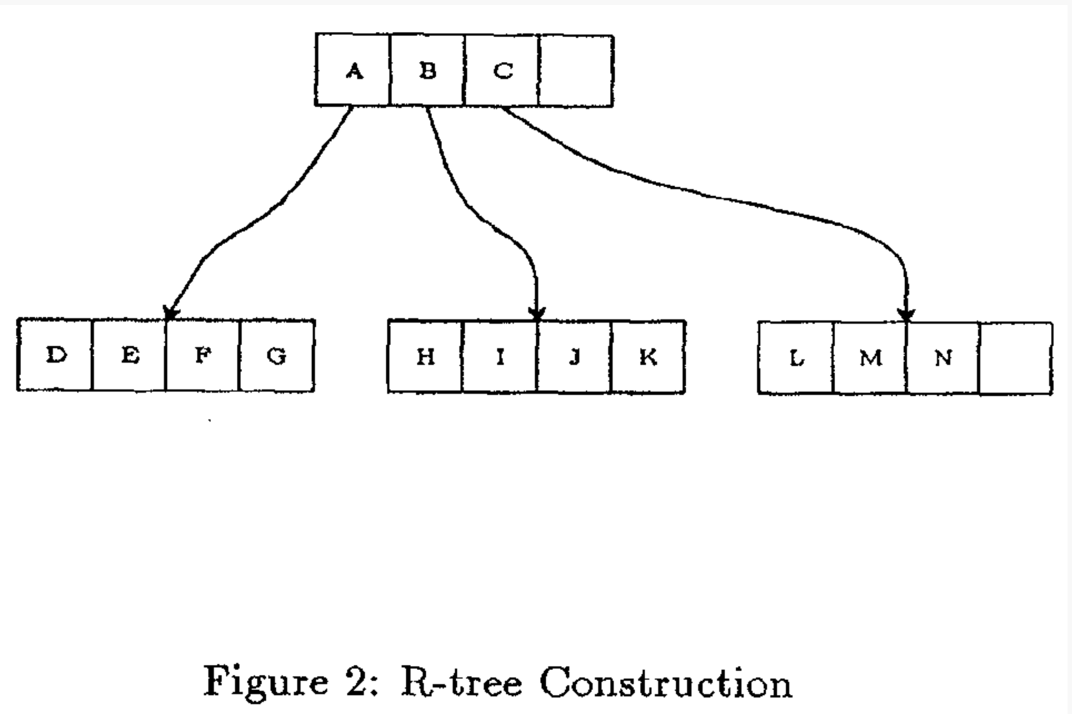


Figure 2: R-tree Construction

Metrics for k-NN Search

- Fan-out (branching factor)
  - The maximum number of entries that a node can have
  - Performance of an R-tree search is measured by the number of disk accesses (reads) necessary to find (or not found) the desired objects in the database.

- Two metrics for ordering the k-NN search
  - Given a query point P and an object O enclosed in its MBR
  1. The minimum distance (MINDIST) of the object O from P
  2. The minimum of the maximum possible distance (MINMAXDIST) from P to a face (or vertex) of the MBR containing O.
  - ⇨ Offer a lower and an upper bound on the actual distance of O from P respectively.
  - ⇨ Used to order and efficiently prune the paths of the search space in an R-tree.

Minimum Distance (MINDIST) - 1/2

- Def. 1. A Rectangle R in Euclidean space E(n) of dimension n, is defined by the 2 endpoints S and T of its major diagonal:

$$R = (S, T)$$

$$\text{where } S = \{s_1, s_2, \ldots, s_n\} \text{ and } T = \{t_1, t_2, \ldots, t_n\}$$

$$\text{and } s_i \le t_i \text{ for } 1 \le i \le n$$

- Def. 2. MINDIST(P, R)

  The distance of a point P in E(n) from a rectangle R in the same space, denoted MINDIST(P,R), is:

$$MINDIST(P,R) = \sum_{i=1}^{n} |p_i - r_i|^2, \quad \text{where} \quad r_i = \begin{cases} s_i & \text{if } p_i < s_i \\ t_i & \text{if } p_i > s_i \\ p_i & \text{otherwise} \end{cases}$$

  If the point is inside the rectangle, the distance between them is 0. If the point is outside, we use the square of the Euclidean distance between the point and the nearest edge of the rectangle.

Minimum Distance (MINDIST) - 2/2

- Lemma 1. The distance of Def. 2 is equal to the square of the minimal Euclidean distance from P to any point on the perimeter of R.
- Def. 2. The minimum distance of a point P from a spatial object o, denoted by ||(P,o)|| is:

$$\|(P,o)\| = min\,(\sum_{i=1}^{n} |p_i - x_i|^2,\ \forall X = [x_1, \ldots, x_n] \in O)$$

- Theorem 1. Given a point P and an MBR R enclosing a set of objects $O = \{o_i, 1 \leq i \leq m\}$, the following is true:

$$\forall o \in O,\ MINDIST(P,R) \leq \|(P,o)\|$$

- Lemma 2. The MBR Face Property

  Every face (i.e., edge in dimension 2, rectangle in dimension 3 and hyperspace in higher dimensions) of any MBR (at any level of the R-tree) contains at least one point of some spatial object in the DB (See Figs. 3 and 4).

MBR Face Property

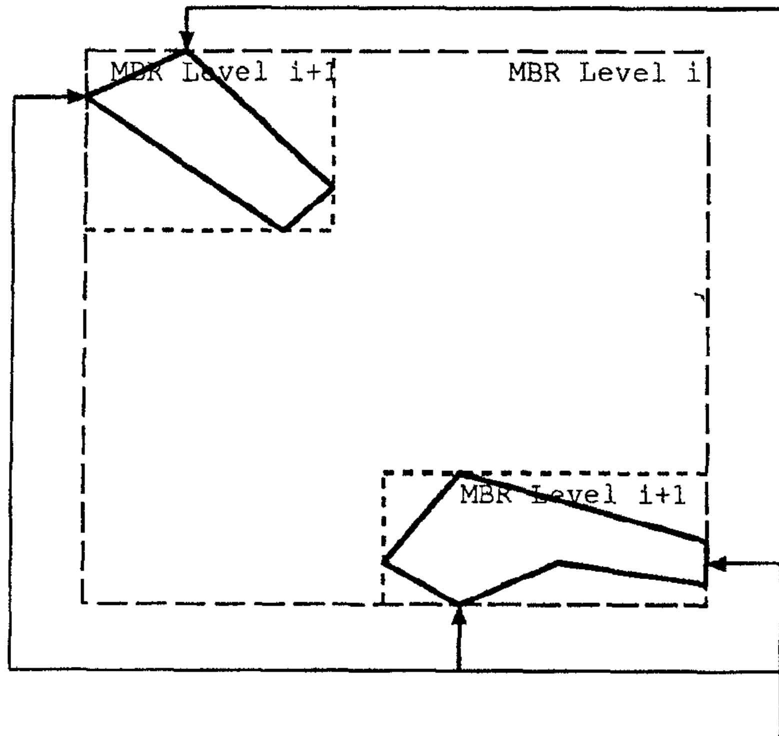


Figure 3: MBR Face Property in 2-Space

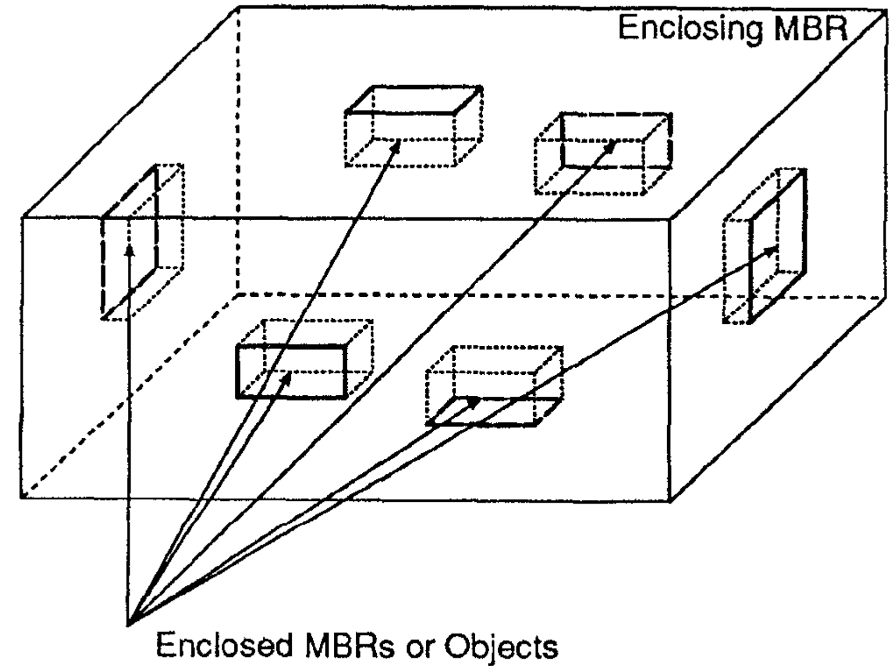


Figure 4: MBR Face Property in 3-Space

Minimax Distance (MINMAXDIST) - 1/2

- Upper bound (MINMAXDIST) of the NN distance to any object inside an MBR
  - Minimum value of all the maximum distances between the query point and points on the each of the n axes
  - Allows to prune MBRs that have MINDIST › upper bound
  - Guarantees there is an object within the distance(MBR) ≤ MINMAXDIST.

Definition 4 *Given a point P in E(n) and an MBR R = (S, T) of the same dimensionality, we define MINMAXDIST(P,R) as:*

$$MINMAXDIST(P, R) =$$

$$\min_{1 \le k \le n} \left(|p_k - rm_k|^2 + \sum_{\substack{i \ne k \\ 1 \le i \le n}} |p_i - rM_i|^2 \right)$$

where:

$$rm_k = \begin{cases} s_k & \text{if } p_k \le \frac{(s_k + t_k)}{2}; \\ t_k & \text{otherwise.} \end{cases} \quad \text{and}$$

$$rM_i = \begin{cases} s_i & \text{if } p_i \ge \frac{(s_i + t_i)}{2}; \\ t_i & \text{otherwise.} \end{cases}$$

MINMAXDIST – 2/2

- Theorem 2. Given a point P and an MBR R enclosing a set of objects $O = \{o_i, 1 \leq i \leq m\}$, the following property holds:

$$\exists \in O, \|(P, o)\| \leq MINMAXDIST(P,R)$$

⇨ Guarantees the presence of an object O in R whose distance from P is within this distance.

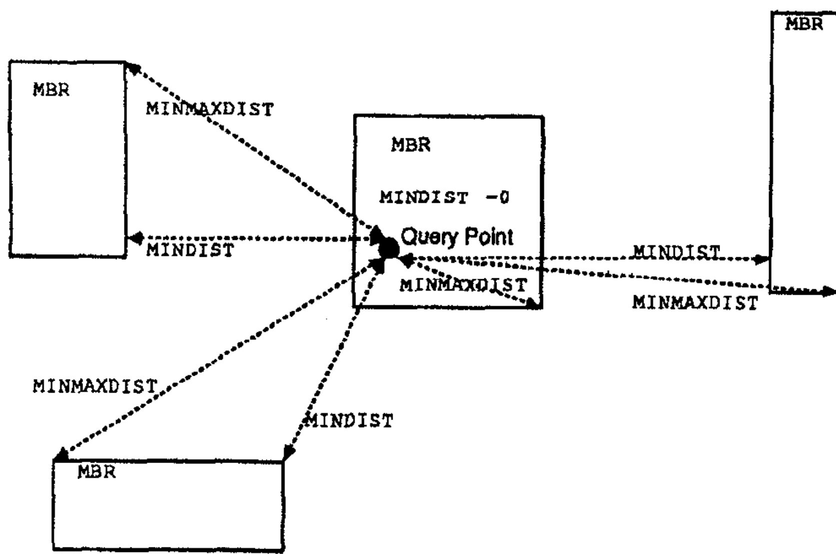


Figure 5: MINDIST and MINMAXDIST in 2-Space

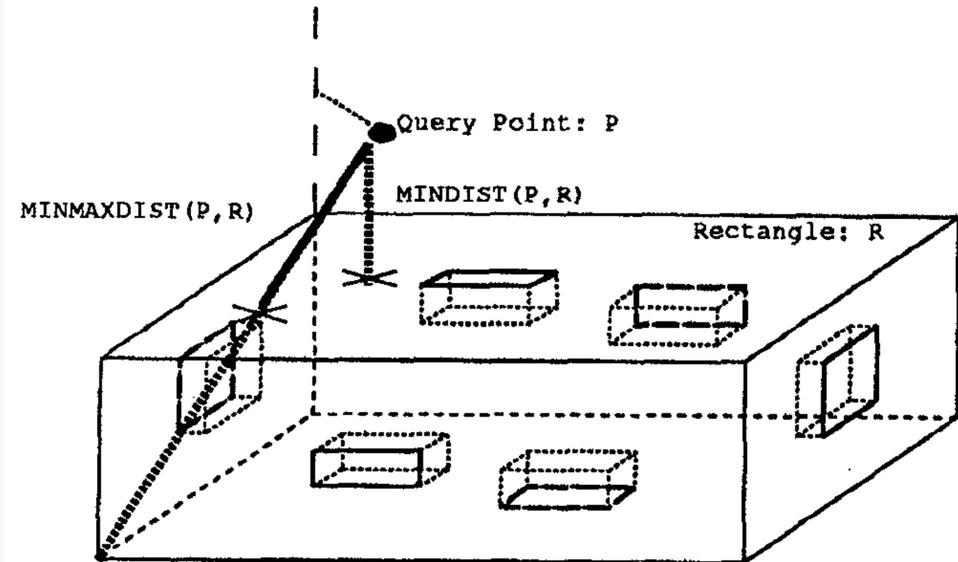


Figure 6: MINDIST and MINMAXDIST 3-Space

MINDIST and MINMAXDIST

- Search Ordering
  - MINDIST ordering is the optimistic choice.
  - MINMAXDIST metric is the pessimistic one.
  - MINDIST metric ordering is not always the best choice.

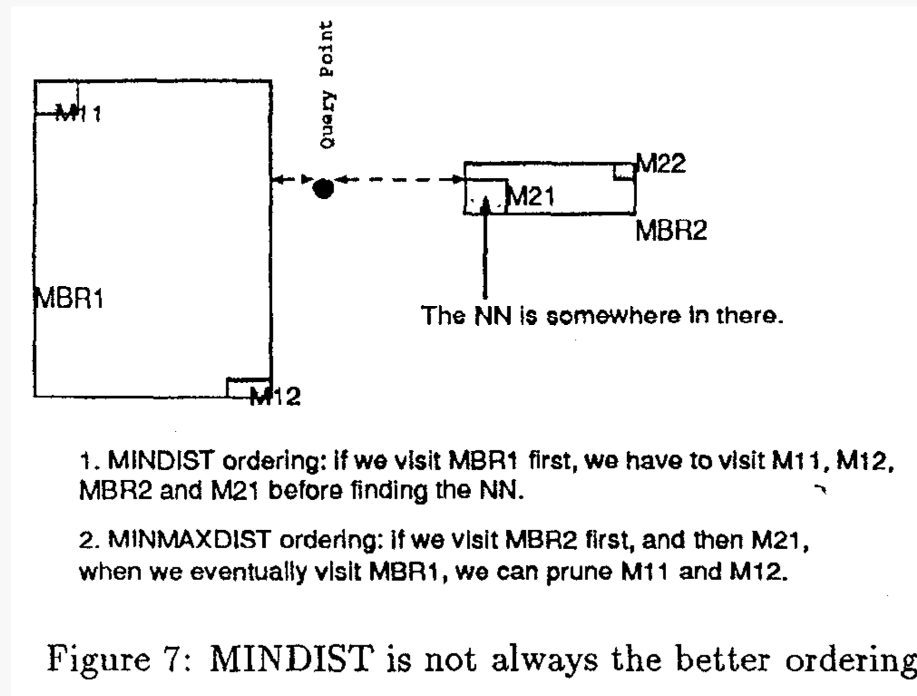


Figure 7: MINDIST is not always the better ordering

Strategies for Search Pruning

1. An MBR M with MINDIST(P,M) greater than the MINMAXDIST(P,M') of another MBR M' is discarded because it cannot contain the NN.

2. An actual distance from P to a given object O which is greater than the MINMAXDIST(P,M) for an MBR M can be discarded because M contains an object O' which is nearer to P.

3. Every MBR M with MINDIST(P,M) greater than the actual distance from P to a given object O is discarded because it cannot enclose an object nearer than O.

Nearest Neighbor Search Algorithm (1/2)

- Ordered depth first traversal

1. Begins with the R-tree root node and proceeds down the tree.
2. During the descending phase, at each newly visited nonleaf node, computes the ordering metric bounds (MINDIST/MAXDIST) for all its MBRs and sorts them into an Active Branch List.
3. Applies pruning strategies to the ABL to remove unnecessary branches.
4. The algorithm iterates on this ABL until the ABL empty: For each iteration, the algorithm selects the next branch in the list and applies itself recursively to the node corresponding to the MBR of this branch.
5. At a leaf node (DB objects level), the algorithm calls a type specific distance function for each object and selects the smaller distance between current value of Nearest and each computed value and updates Nearest appropriately.
6. At the return from the recursion, we take this new estimate of the NN and apply pruning strategy 3 to remove all branches with MINDIST(P,M) > Nearest for all MBRs M in the ABL.

Nearest Neighbor Search Algorithm (2/2)

```
RECURSIVE PROCEDURE
nearestNeighborSearch (Node, Point, Nearest)

NODE        Node        // Current NODE
POINT       Point       // Search POINT
NEARESTN    Nearest     // Nearest Neighbor

    //Local Variables
    NODE            newNode
    BRANCHARRAY     branchList
    integer         dist, last, i

    // At leaf level - compute distance to actual objects
    If Node.type = LEAF
    Then
        For i := 1 to Node.count
            dist := objectDIST(Point, Node.branch_i.rect)
            If (dist < Nearest.dist)
                Nearest.dist := dist
                Nearest.rect := Node.branch_i.rect

    // Non-leaf level - order, prune and visit nodes
Else
    // Generate Active Branch List
    genBranchList(Point, Node, branchList)

    // Sort ABL based on ordering metric values
    sortBranchList(branchList)

    // Perform Downward Pruning
    // (may discard all branches)
    last = pruneBranchList(Node, Point, Nearest,
            branchList)

    // Iterate through the Active Branch List
    For i := 1 to last
        newNode := Node.branch_branchList_i

        // Recursively visit child nodes
        nearestNeighborSearch(newNode, Point,
            Nearest)

        // Perform Upward Pruning
        last := pruneBranchList(Node, Point, Nearest,
            branchList)
```

Generalization: Finding *k* NNs

- The differences from 1-NN algorithm are:
  - A sorted buffer of at most *k* current NNs is needed.
  - The MBRs pruning is done according to the distance of the farthest NN in this buffer.

Experimental Results (1/2)

➢ TIGER data files for Long Brach and Montgomery.

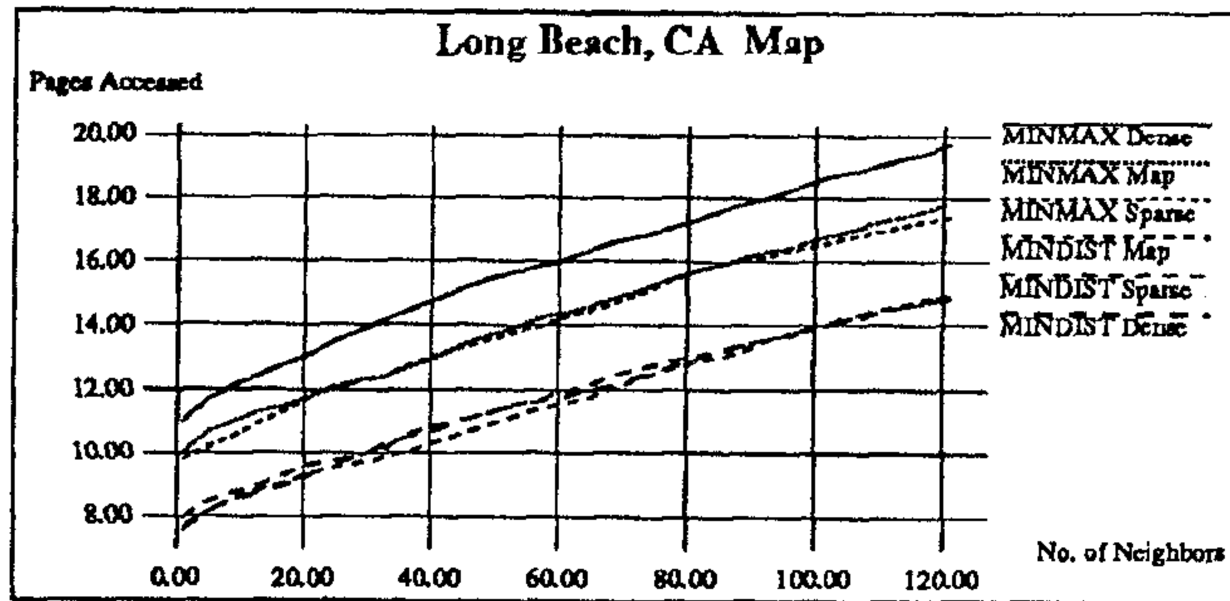


Figure 9: MINDIST and MINMAXDIST Metric Comparison

Experimental Results (2/2)

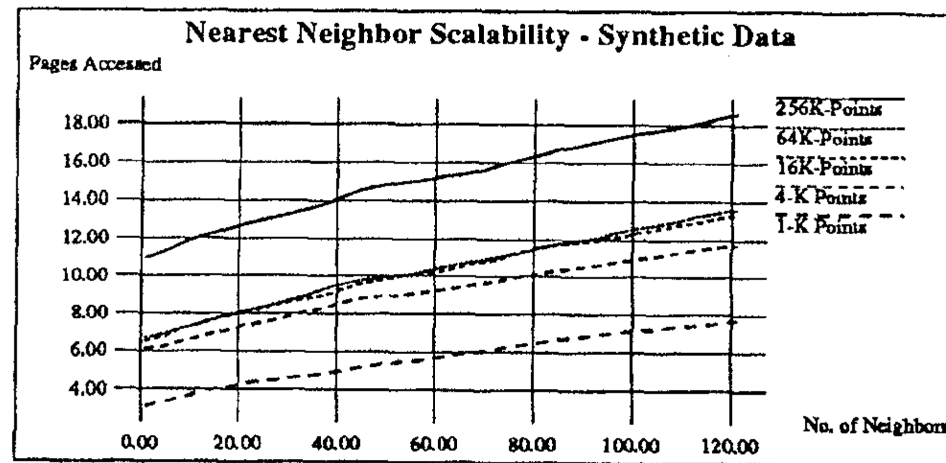


Figure 10: Synthetic Data Experiment 1 Results

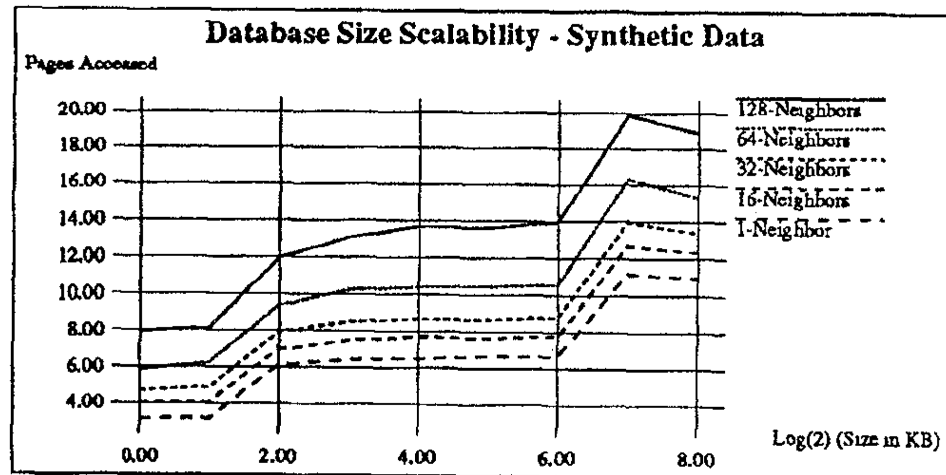


Figure 11: Synthetic Data Experiment 2 Results

Synopsis

We have discussed:

- The R-tree and R*-tree
- The X-tree
- The Pyramid technique
- An algorithm for k-NN search

There are other methods such as:

TV-tree, SS-tree, SR-tree, VA-File.

Bibliography

+++