THE R+-TREE: A DYNAMIC INDEX FOR MULTI-DIMENSIONAL OBJECTS

Timos Sellis (*University of Maryland - College Park)*

Nick Roussopoulos (*University of Maryland - College Park)*

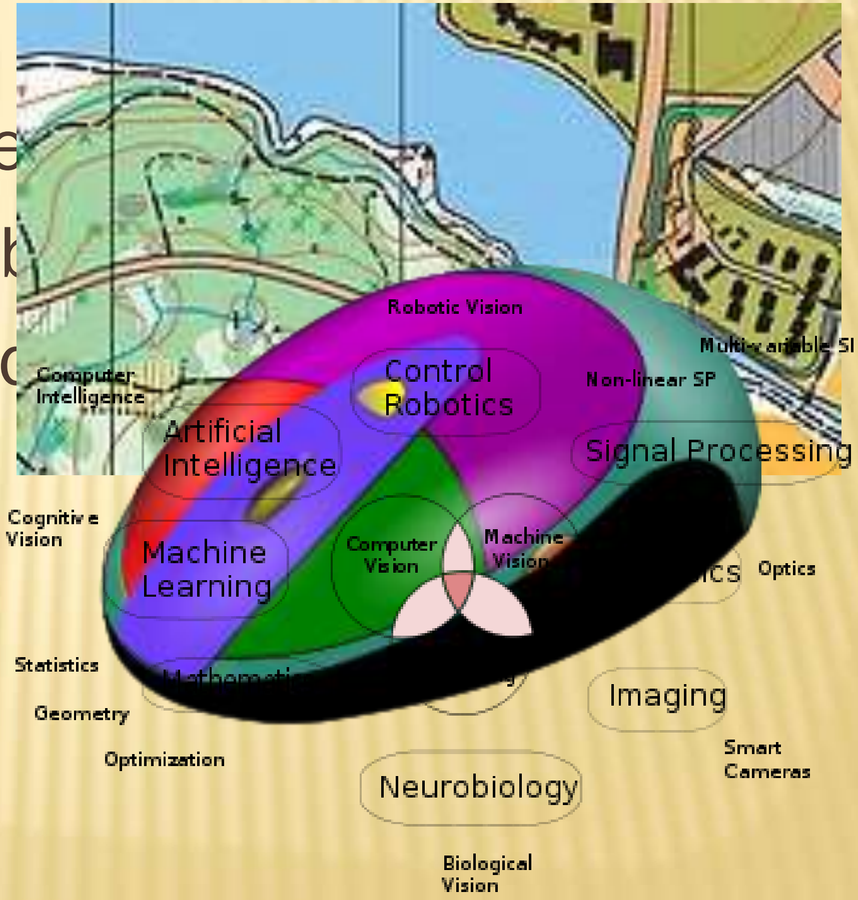
Christos Faloutsos (*Carnegie Mellon University)*

Presenter: Xunfei Jiang

INTRODUCTION

- Data Categories
  - One-dimensional data
    - Integer
    - Real numbers
    - Strings
  - Multi-dimensional data
    - Boxes
    - Polygons
    - Points in multi-dimensional space

- Multi-dimensional data in application areas
  - Cartography
  - CAD(Computer-Aided De
  - Computer Vision and rob
  - Rule indexing in expert c



- DBMS with multi-dimensional data
  - Addressed operations
    - Point queries
      - Given a point in the space, find all objects that contain it
    - Region queries
      - Given a region (query window), find all objects that intersect it
  - Un- addressed operations
    - Insertion
    - Deletion
    - modification

Need support in dynamic environment

SURVEY

- Classification of multi-dimensional objects
  - Points
  - Rectangles
    - Circles, polygons and other complex objects can be reduced to rectangles(MBRs)

POINTS

- Method
  - divide the whole space into **disjoint** sub-regions
    - each sub-region contains no more than *C* points
    - usually *C* = 1 /the capacity of a disk page(number of data records the page can hold)
- Operations
  - Insertion
    - Split: further partition of a region
      - introduce a hyper-plane and divided region into disjoint sub-regions

- Attribute of Split
  - Pos
    - F
    - A … r-plane
  - Dim
    - C
    - K hyper-plane
  - Locality
    - Grid method: split all regions in this direction
    - Brickwall method: split only the region that need to be spitted

Method	Position	Dimensions	Locality
point quad-tree	adaptable	k-d	brickwall
k-d tree	adaptable	1-d	brickwall
grid file	fixed	1-d	grid
K-D-B-tree	adaptable	1-d	brickwall

Table 2.1: Illustration of the classification.

RECTANGLES

- Methods classification
  - (1) transform the rectangles into points in a space of higher dimensionality
    - Eg: 2-d rectangle be considered as 4-d point
  - (2) use *space filling curves* to map a k-d space onto a 1-d space
    - Eg: transform k-dimensional objects to line segments, using the so-called *z-transform.*
    - preserve the distance
      - points that are close in the k-d space are likely to be close in the 1-d transformed space

- (3) divide the original space into appropriate sub-regions
  - **Disjoint regions:** any of the methods for points could be used for rectangles
    - rectangle intersect a splitting hyper-plane
      - Solution: cut the offending rectangle in two pieces and tag the pieces, to indicate that they belong to the same rectangle.
      - Splitting hyper-planes can be of arbitrary orientation(not necessarily parallel to the axes).
  - **Overlapping regions:**
    - Guttman proposed R-Trees
      - extension of B-trees for multi-dimensional objects that are either points or regions.
      - Guarantee that the space utilization is at least 50%.
      - if R-Trees are built using the dynamic insertion algorithms, the structure may provide excessive space overlap and "dead-space" in the nodes that result in bad performance. (R+-tree address this problem)

R TREE

- R-tree
  - Extension of B-tree in k-dimensions
  - Height-balanced tree
  - Components
    - Intermediate nodes: grouping rectangles
    - leaf nodes: data objects

Each intermediate node encloses all rectangles that are correspond to lower level nodes

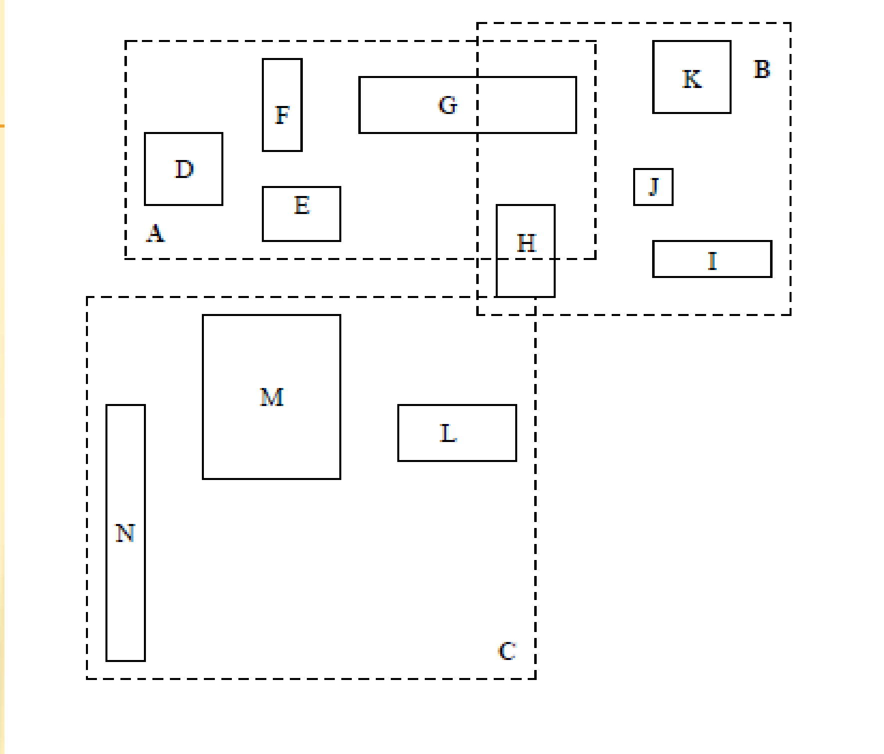


Figure 3.1: Some rectangles organized into an R-tree

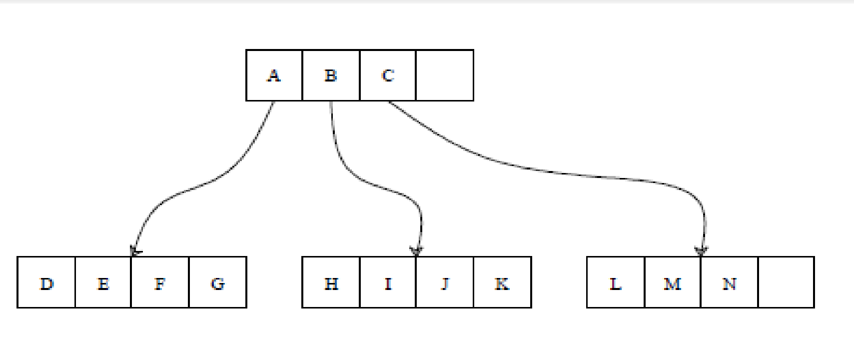


Figure 3.2: R-tree for the rectangles of Figure 3.1

R-TREE

- Coverage
  - The total area of all the rectangles associated with the nodes of that level.
- Overlap
  - the total area contained within two or more nodes.

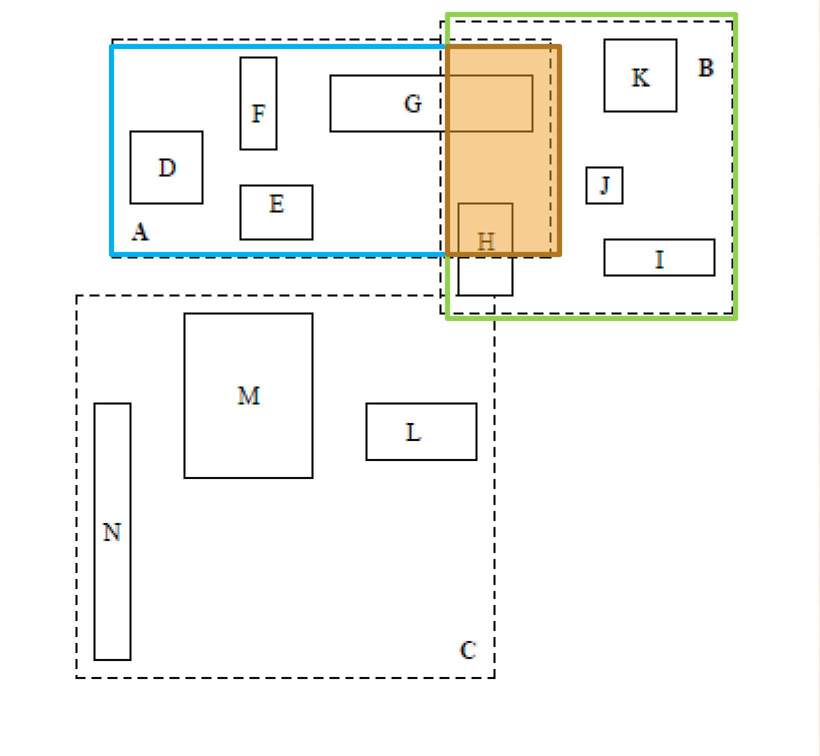


Figure 3.1: Some rectangles organized into an R-tree

- Efficient R-tree
  - Minimize coverage
    - reduce dead space(i.e. empty space)
  - Minimize overlap
    - E.g: search window w result in search both nodes A and B
- Zero overlap & coverage?
  - Achievable for data points that are known in advance
  - Zero overlap is not attainable for region objects

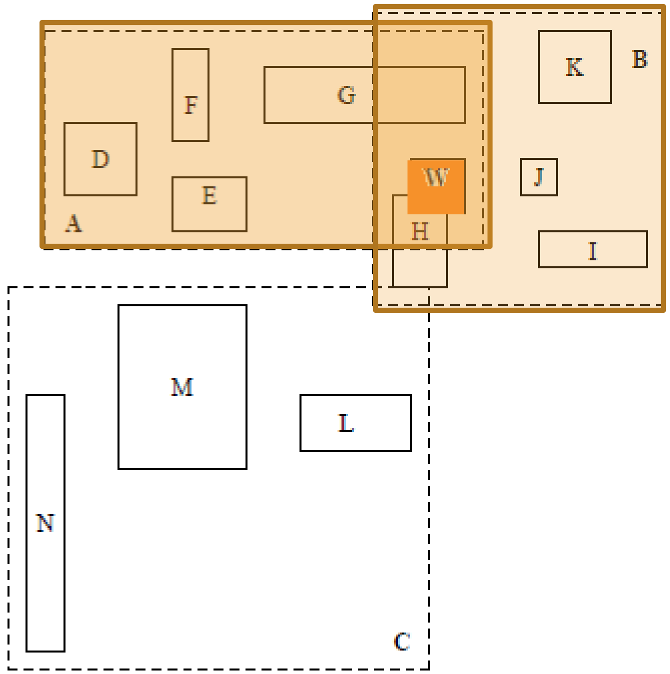


Figure 3.3: An example of a "bad" search window

partition

Zero overlap

R+ Tree

R+ TREE

- Whenever a data rectangle at a lower level overlaps with another rectangle, decompose it into two non-overlapping sub-rectangles
  - Eg: Rectangle G is split into two sub-rectangles: one contained in node A; the other contained in node P.
- Pros and cons:
  - time saving on searching
  - increase space cost

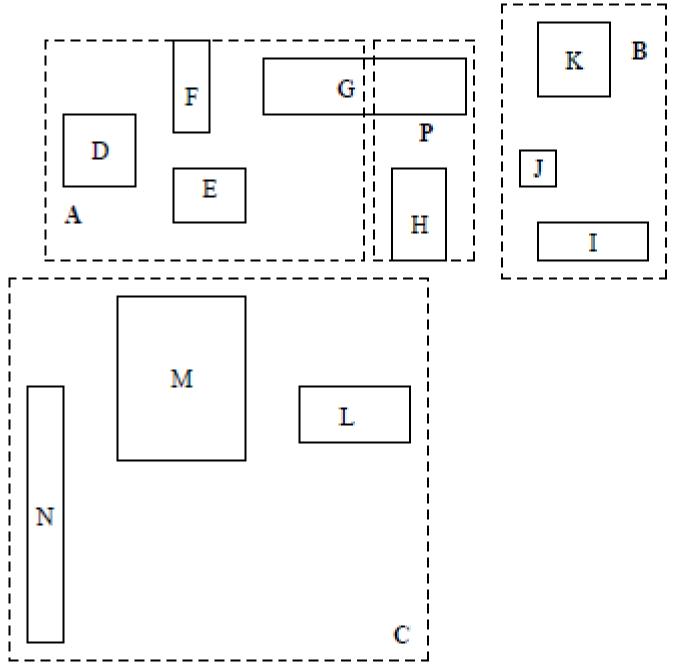


Figure 3.4: The rectangles of Figure 3.1

R+ TREE

- Structure

Leaf node

(oid, RECT)

Data object

Bounds of data objects

$$\left(x_{low}, x_{high}, y_{low}, y_{high}\right)$$

Intermediate node

(p, RECT)

Pointer to a lower level node

Rectangle that encloses the nodes in lower level

R+ TREE

- Properties
  - (1) For each entry (*p, RECT) in an intermediate* node, the sub-tree rooted at the node pointed to by *p contains a rectangle R if and only if R is* covered by *RECT.*
    - *Exception: R is a rectangle at a leaf node* -> *R* must just overlap with *RECT.*
  - (2) For any two entries (p_1,$RECT_1$) *and* (p_2,$RECT_2$) *of an intermediate node, the overlap* between $RECT_1$ *and* $RECT_2$ *is zero.*
  - (3) The root has at least two children unless it is a leaf.
  - (4) All leaves are at the same level.

SEARCH

Search(R,W) ⟶ Search(P,W) ⟶ Search(H,W) ⟶ H

Algorithm Search (R, W)

Input:

An R$^+$-tree rooted at node R and a search window (rectangle) W

Output:

All data objects overlapping W

Method:

Decompose search space and recursively search tree

S1. [Search Intermediate Nodes]

If R is not a leaf, then for each entry (p, $RECT$) of R check if $RECT$ overlaps W. If so, **Search**($CHILD, W \cap RECT$), where $CHILD$ is the node pointed to by p.

S2. [Search Leaf Nodes]

If R is a leaf, check all objects $RECT$ in R and return those that overlap with W.

Figure 3.6: Searching algorithm

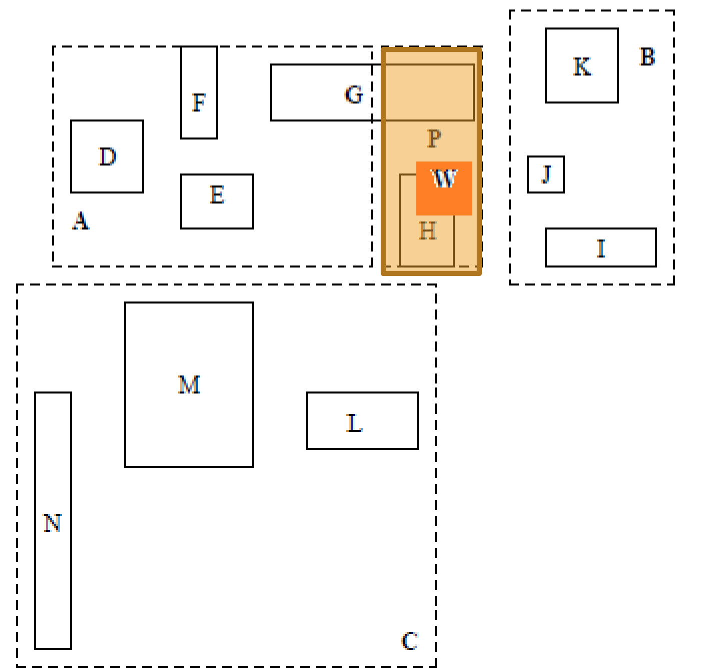


Figure 3.4: The rectangles of Figure 3.1

INSERT

Algorithm Insert (*R*,*IR*)

Input:

An R$^+$-tree rooted at node *R* and an input rectangle *IR*

Output:

The new R$^+$-tree that results after the insertion of *IR*

Method:

Find where *IR* should go and add it to the corresponding leaf nodes

I1. [Search Intermediate Nodes]
If *R* is not a leaf, then for each entry (*p*, *RECT*) of *R* check if *RECT* overlaps *IR*. If so, **Insert**(*CHILD*,*IR*), where *CHILD* is the node pointed to by *p*.

I2. [Insert into Leaf Nodes]
If *R* is a leaf, add *IR* in *R*. If after the new rectangle is inserted *R* has more than *M* entries, **SplitNode**(*R*) to re-organize the tree (see section 3.5).

Figure 3.7: Insertion algorithm

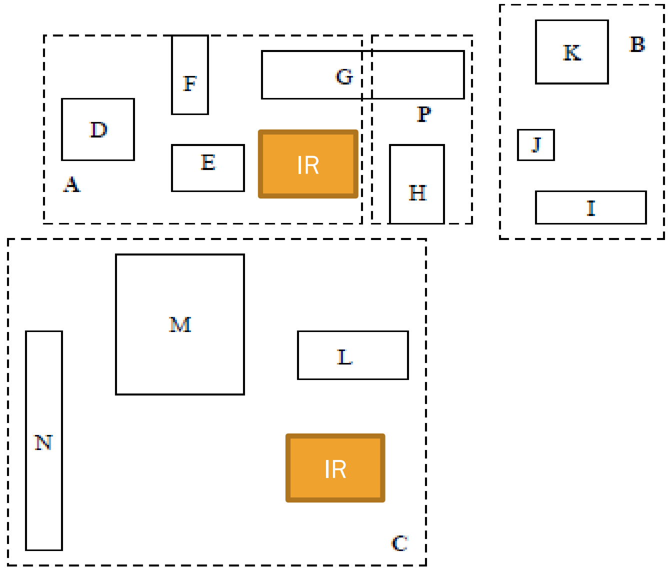
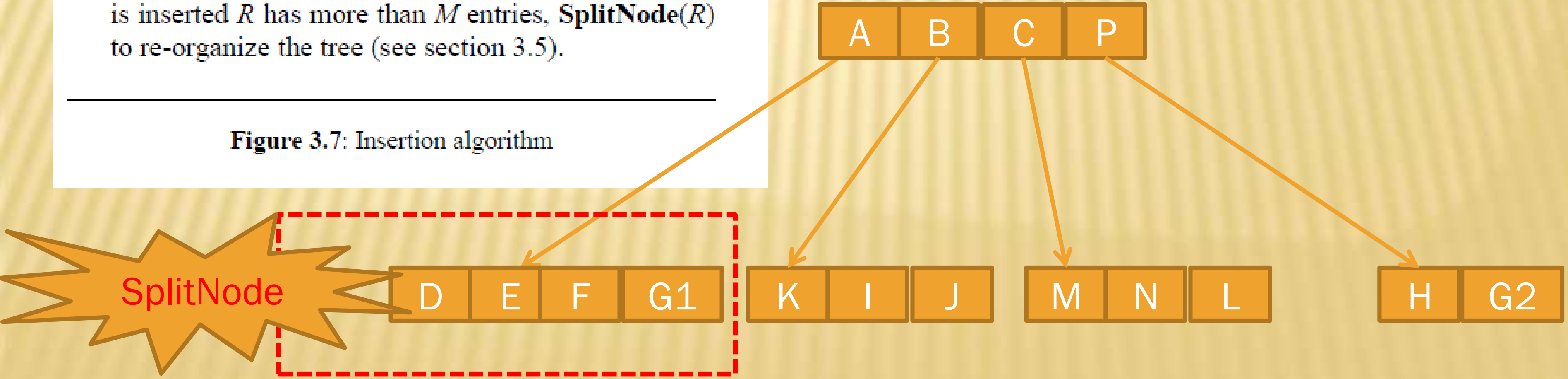


Figure 3.4: The rectangles of Figure 3.1



DELETION

Algorithm Delete (*R*,*IR*)

Input:

An R^+-tree rooted at node *R* and an input rectangle *IR*

Output:

The new R^+-tree that results after the deletion of *IR*

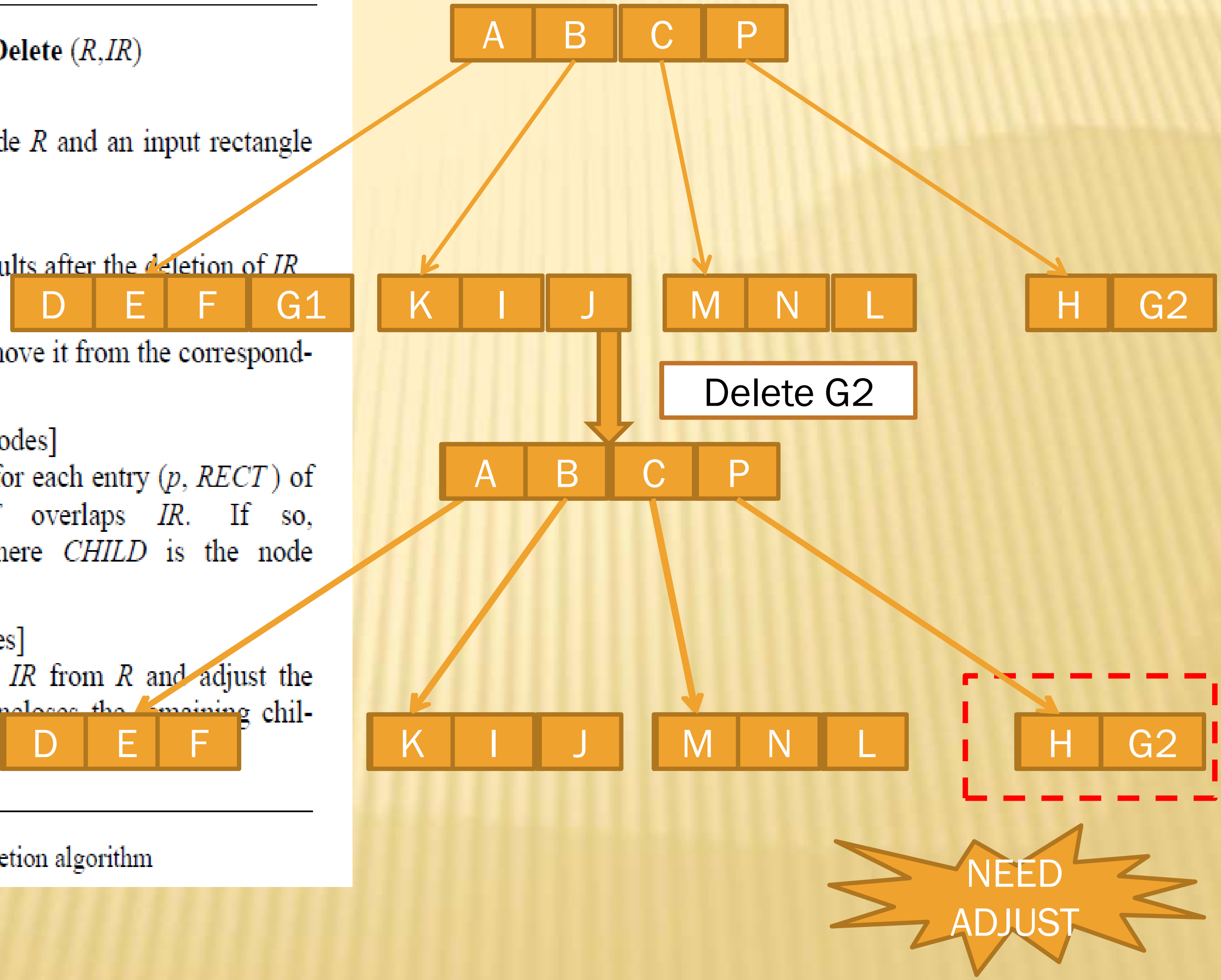
Method:

Find where *IR* is and remove it from the corresponding leaf nodes.

D1. [Search Intermediate Nodes]
If *R* is not a leaf, then for each entry (*p*, *RECT*) of *R* check if *RECT* overlaps *IR*. If so, **Delete**(*CHILD*,*IR*), where *CHILD* is the node pointed to by *p*.

D2. [Delete from Leaf Nodes]
If *R* is a leaf, remove *IR* from *R* and adjust the parent rectangle that encloses the remaining children rectangles.

Figure 3.8: Deletion algorithm



NODE SPLITTING

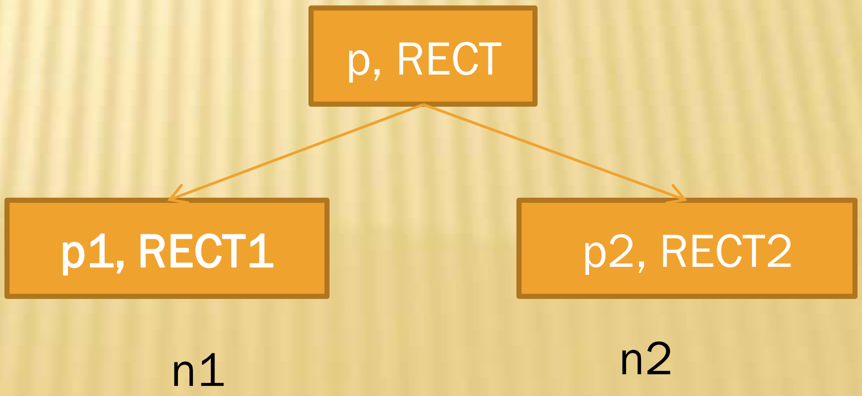
Input: A node *R* (leaf or intermediate)

Output: The new R+-tree

Method: **[SN1]Find a partition for the node to be split, [SN2]create two new nodes and, if needed, [SN3]propagate the split upward and downward**

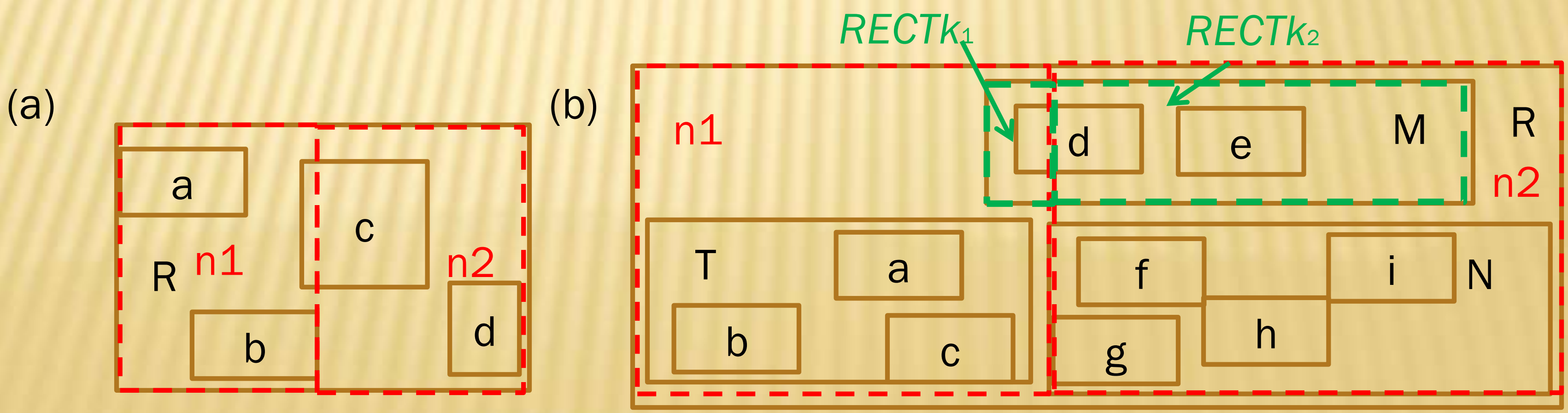
+ SN1. [Find a Partition]

- Partition *R* using the **Partition** routine of the **Pack** algorithm (see next section).
- Partition node R (p, RECT), let *S*1 and *S*2 denote the two sub-regions resulting after the partition. Create two nodes:
  - *n*1=(*p*1,*RECT*1)
  - *n*2=(*p*2,*RECT*2)
  - *RECTi*=*RECT* ∩ *Si* (*i*=1,2)



NODE SPLITTING

- SN2. [Populate New Nodes]
  - Put all the sub-nodes of R into n_i (i = 1,2)
  - For those nodes(pk, RECTk) that overlap with the sub-regions
    - a) *R* is a leaf node, put *RECTk* in both new nodes
    - b) Otherwise, use **SplitNode** to recursively split the children nodes along the partition.
      - Let (pk_1,$RECTk_1$) and (pk_2,$RECTk_2$) be the two nodes after splitting (*pk*,*RECTk*), where *RECTki* lies completely in *RECTi* , *i*=1,2.
      - Add those two nodes to the corresponding node *ni* .



NODE SPLITTING

- SN3. [Propagate Node Split Upward]
  - If *R* is the root, create a new root with only two children, *n*1 and *n*2.
  - Otherwise, let *PR* be *R*'s parent node. Replace *R* in *PR* with *n*1 and *n*2. If *PR* has now more than *M* entries, invoke **SplitNode**(*PR*).

PACKING ALGORITHM

- Partition
  - divides the total space occupied by *N* 2-dimensional rectangles by a line parallel to the *x*-axis(*x*_cut) or the *y*-axis (*y*_cut).
    - The selection of the *x*_cut or *y*_cut is based on one or more of the following criterias:
      - (1) nearest neighbors
      - (2) minimal total *x* − and *y*-displacement
      - (3) minimal total space coverage accrued by the two sub-regions
      - (4) minimal number of rectangle splits.

(1)(2)(3) reduce search by reducing the coverage of "dead-space".

(4) confines the height expansion of the R+-tree

PARTITION

Algorithm Partition (*S*,*ff*)

Input:
A set of *S* rectangles and the fill-factor *ff* of the first sub-region

Output:
A node *R* containing the rectangles of the first sub-region and the set *S'* of the remaining rectangles

Method:
Decompose the total space into a locally optimal (in terms of search performance) first sub-region and the remaining sub-region

PA1. [No Partition Required]
If total space to be partitioned contains less than or equal to *ff* rectangles, no further decomposition is done; a node *R* storing the entries is created and the algorithm returns (*R*,*empty*).

PA2. [Compute Lowest *x*- and *y*- Values]
Let *Ox* and *Oy* be the lowest *x*- and *y*-coordinates of the given rectangles.

PA3. [Sweep Along the *x*-dimension]
(*Cx*,*x_cut*) = **Sweep**("x",*Ox*,*ff*). *Cx* is the cost to split on the *x* direction.

PA4. [Sweep Along the *y*-dimension]
(*Cy*,*y_cut*) = **Sweep**("y",*Oy*,*ff*). *Cy* is the cost to split on the *y* direction.

PA5. [Choose a Partition Point]
Select the cut that gives the smallest of *Cx* and *Cy*, divide the space, and distribute the rectangles and their splits. A node *R* that stores all the entries of the first sub-region is created. Let *S'* denote the set of the rectangles falling in the second sub-region. Return (*R*,*S'*).

Figure 4.1: Partition algorithm

Algorithm Sweep (*axis*,*Oxy*,*ff*)

Input:
The axis on which sweeping is performed, the point *Oxy* on that axis where the sweep starts and the fill-factor *ff*

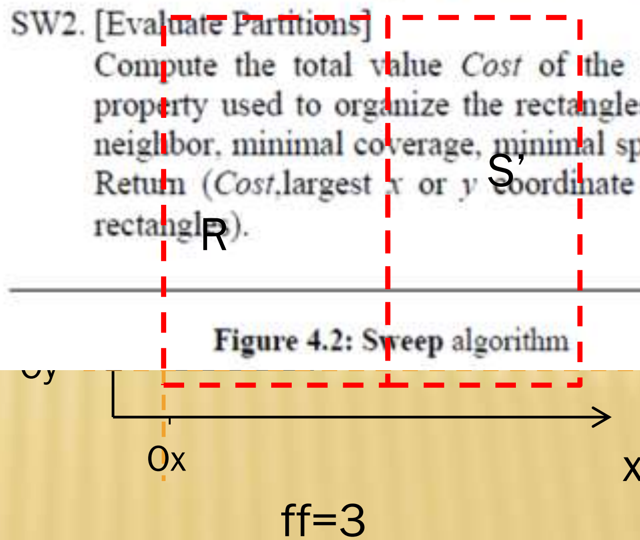
Output:
Computed properties of the first sub-region and the *x*_ or *y*_cut

Method:
Sweep from *Oxy* and compute the property until the *ff* has been reached

SW1. [Find the First *ff* Rectangles]
Starting from *Oxy*, pick the next *ff* rectangles from the list of rectangles sorted on the input axis.

SW2. [Evaluate Partitions]
Compute the total value *Cost* of the measured property used to organize the rectangles (nearest neighbor, minimal coverage, minimal spilts, etc.). Return (*Cost*,largest *x* or *y* coordinate of the *ff* rectangles).

Figure 4.2: Sweep algorithm



PACK

Algorithm Pack (S, ff)

Input:
A set S of rectangles to be organized and the fill-factor ff of the tree

Output:
A "good" R^+-tree

Method:
Recursively pack the entries of each level of the tree

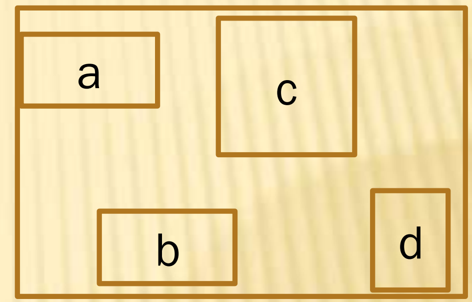
P1. [No Packing Needed]
If $N=|S|$ is less than or equal to ff, then build the root R of the R^+-tree and return it.

P2. [Initialization]
Set AN=*empty*. AN holds the set of next level rectangles to be packed later.

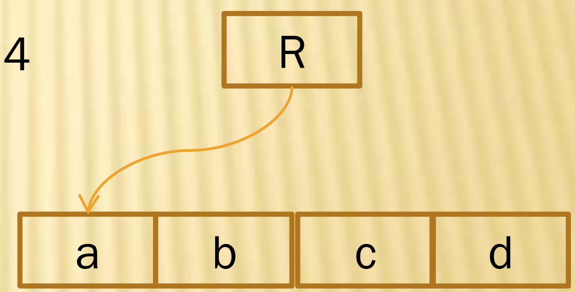
P3. [Partition Space]
$(R,S') =$ **Partition**(S, ff)
if we are partitioning non-leaf nodes and some of the rectangles have been split because of the chosen partition, recursively propagate the split downward and if necessary propagate the changes upward also.
AN=append(AN,R).
Continue step P3 until S' = *empty*.

P4. [Recursively Pack Intermediate Nodes]
Return **Pack**(AN, ff)

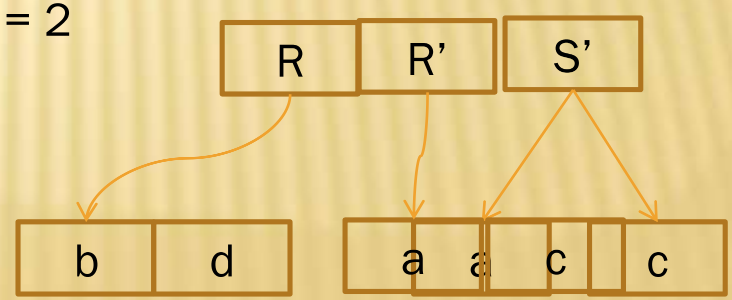
Figure 4.3: Pack algorithm



ff = 4



ff = 2



AN = {R}

AN = {R,R'}

ANALYSIS

- Rectangle
  - 4 coordinates are enough to uniquely determine it (the x and y coordinates of the lower-left and upper-right corners).
  - examine segments on a line (1-d space) instead of rectangles in the plane (2-d space), and transform the segments into points in a 2-d space.
    - Each segment is uniquely determined by (*xstart* , *xend*), the coordinates of its start and end points.
    - *Density(D)*
      - the number of segments that contain a given point

SEARCH PERFORMANCE IN QUERY OF POINTS

100,000 segments

total density: 40

- Figure 5.1a
- disk accesses=f(large segment density)
  - large segments account for 10% of the total number of segments
  - N1=90,000
  - N2=10,000
- Figure 5.1b
- disk accesses= f(small segments)
  - small segment density (D1=5).

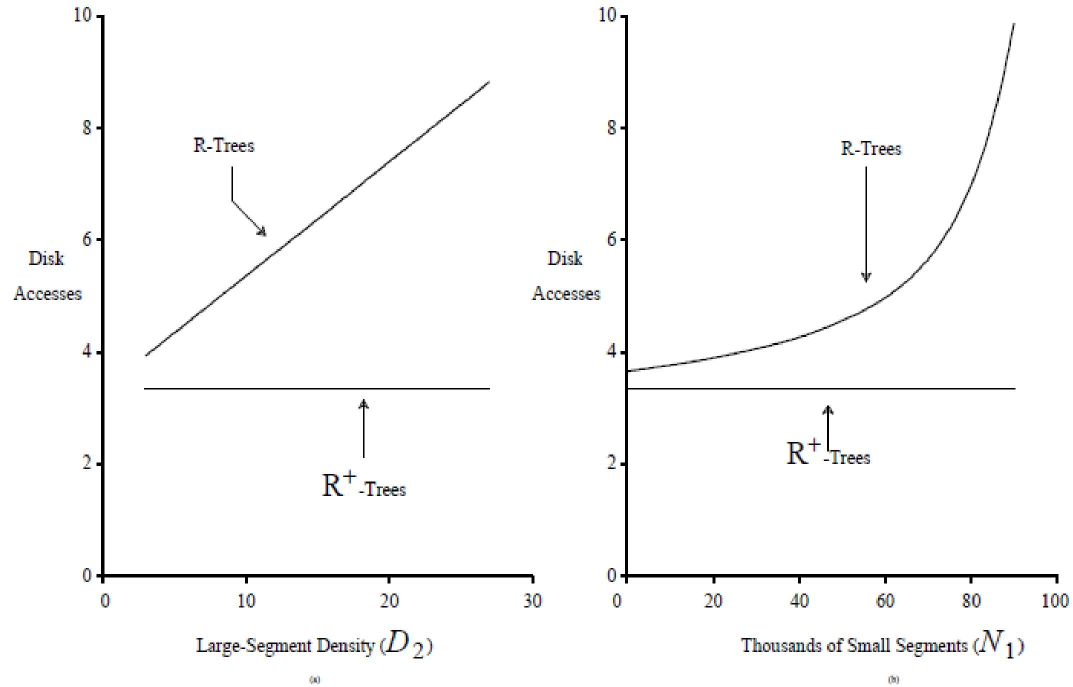


Figure 5.1

Disk Accesses for Two-Size Segments: *Point Query*

(a) As a function of D_2; N_2=10,000

(b) As a function of N_1; D_1=5

SEARCH PERFORMANCE IN QUERY OF SEGEMENTS

- N1 increase, few lengthy segments :
  - R+-trees gain a performance improvements of up to 50%.
- N2 approaches the total number of segments, R+-trees will lose
  - many lengthy segments cause a lot of splits to sub-segments.

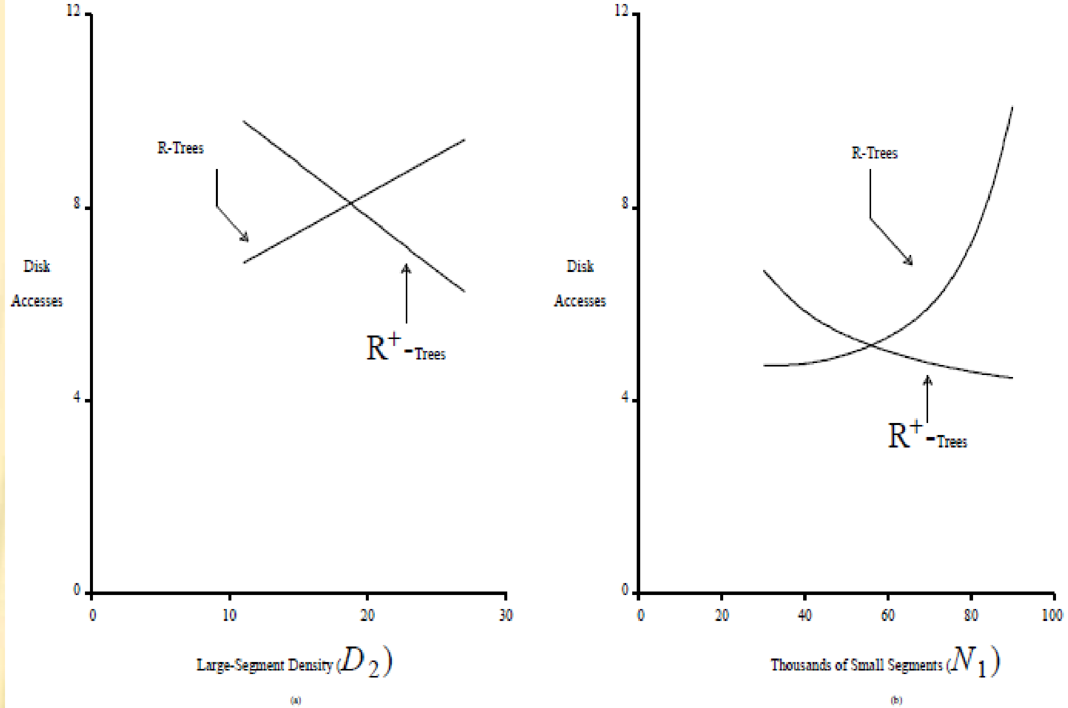


Figure 5.2

Disk Accesses for Two-Size Segments: *Segment Query*

(a) As a function of D_2; N_2=10,000

(b) As a function of N_1; D_1=5

CONCLUSION

- Advantage of R+-trees compared to R-trees
  - improve search performance
    - especially in point queries, more than 50% savings in disk accesses.
    - R-trees suffer in the case of few, large data objects
      - force a lot of "forking" during the search.
    - R+-trees handle these cases easily
      - they split these large data objects into smaller ones.
  - behaves exactly as a K-DB-tree(efficient for indexing point data) in the case where the data is points instead of non-zero area objects (rectangles).

FUTURE WORK

- Experimentation through simulation to verify the analytical results.
- Extension of the analysis for rectangles on a plane (2-d), and eventually for spaces of arbitrary dimensionality.
- Design and experimentation with alternative methods for partitioning a node and compacting an R+-tree.
- Comparison of R- and R+-trees with other methods for handling multi-dimensional objects.

Thanks!