Indexing - Motivation

- Once the data records are stored on a disk, how do you search efficiently?
- Brute force: Retrieve all records, report the qualifying ones
- Better: use indices (pointers) to locate the records directly

Index Evaluation Metrics

Indexing techniques evaluated on the basis:

- Type of queries supported efficiently. eg., exact match, range queries.
- Retrieval time (usually in number of I/O's)
- Insertion and Deletion time
- Space overhead

Spatial Databases

- A spatial database is a system that stores and manages spatial objects. eg., points, lines and polygons in two and three dimensions.
- Applications:
  - GIS (maps)
  - CAD (VLSI design)
  - multimedia databases

Spatial Queries

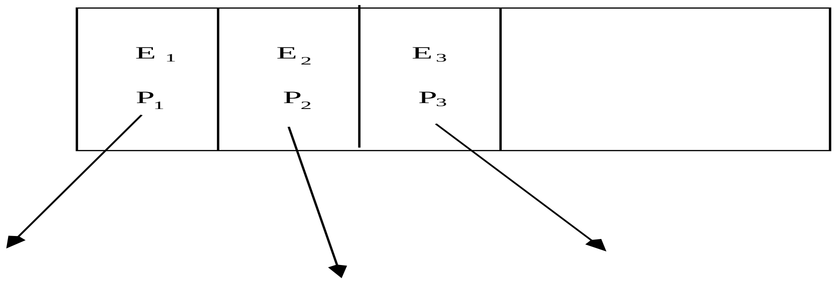
- Point queries
- Range (or window) queries
- Nearest Neighbor queries
- Join queries

Spatial Indexing

- Usually an index stores the MBRs (minimum bounding rectangles) of the indexed objects
- Thus answering a query is a two step process:
  - Filtering step: Find the MBRs that satisfy the query and generate the Candidate Set of Objects
  - Refinement step: For each object in the candidate Set, find the objects that satisfy the query

R-Tree Node

- A node contains between m and M entries
- Each entry has the form: $(E, pointer)$
- If the node is an index node E is the MBR of all objects contained in the subtree pointed by $pointer$.
- If the node is a leaf node, then E is the MBR of the object pointed by $pointer$



R+-Tree

- An improvement over the original R-Tree
- Basic idea: reduce the *coverage* and the *overlap* at each level of the tree
- *Coverage:* total area of all rectangles at a specific level
- *Overlap:* total area contained within two or more nodes
- In R+-tree we try to minimize the coverage, so reduce the *empty* space and allow no overlap

R+-Tree - Properties

The R+-Tree has the following properties:

- For any two entries of an intermediate node, the overlap is zero
- The root has at least two children unless it is a leaf
- All leaves are at the same level

Note: a node is not necessary to contain more than M/2 entries.

R+-Tree - Insertion

INSERT(R,IR)

I1: If R is not a leaf, then for each entry of R check if overlaps IR. If so **INSERT(CHILD, IR)**, where CHILD is the node pointed by the entry

I2: If R is a leaf, add IR in R. If R has more than M entries, call **SplitNode(R)**

R+-Tree - SplitNode

SplitNode(R)

SN1: Partition R using the **Partition**. Create two nodes resulting from the split of R.

SN2: Populate the new nodes. If one entry overlaps with two subregions:

a) If R is a leaf node, then put the entry in both nodes.

b) Otherwise, Use **SplitNode** to recursively split the children nodes along the partition.

SN3: Propagate Node Split Upward. If R is root, create a new root with only two children.

R+-Tree - Partition

Partition(S, ff)

PA1: If total space to be partitioned contains less than ff rectangles, return (R, empty)

PA2: Let Ox and Oy be the lowest x- and y- coordinates of the given rectangles

PA3: Sweep along the x-dimension and find the split. Compute the cost of this split

PA4: Do the same for the y-dimension

PA5: Select the cut that gives the smallest cost, divide the space and distribute the rectangles and their splits. A node R that stores all the entries in the first subregion is created. Let S' be the set of the other rectangles. Return (R, S')

R+-Tree - Pack

Pack(S, ff)

P1: If N=$|S|$ less than or equal to ff, done

P2: Set AN to *empty*. AN holds the set of the next level rectangles

P3: (R, S') = **Partition(S, ff)**
AN = append(AN, R). Propagate splits if necessary. Continue P3 until S' = *empty*

P4: Return **Pack(AN, ff)**

R+-Tree

- Advantages: Faster point searches, and sometimes range searches
- Disadvantages: Splits propagate upward and downward. R+-tree cannot guarantee a minimum space utilization.

R*-Tree

- R*-tree is the most successful variant in the R-tree family
- Optimized insertion algorithm and *forced reinsertion*: if a node overflows, not split but remove p entries and reinsert them into the tree.

R*-Tree - ChooseSubtree

ChooseSubtree

CS1: Set N to be the root

CS2:

```
if N is a leaf,
    return N
else
    if the N points to leaves, then choose node that needs
         least overlap enlargement, then area enlargement,
         then smallest area
    if the N points to non-leaf nodes, choose the node that
         needs least area enlargement, then smallest area
```

CS3: Set N to be the chosen entry and continue from **CS2**.

R*-Tree - Split

Split

S1: Invoke ChooseSplitAxis to determine the axis, perpendicular to which the split is performed

S2: Invoke ChooseSplitIndex to determine the best distribution into two groups along the axis

S3: Distribute the entries into two groups

R*-Tree - ChooseSplitAxis

ChooseSplitAxis

CSA1: For each axis: sort the entries by the lower then by the upper value and determine all distributions. Compute the sum of all margin-values of the different distr. (S)

CSA2: Choose the axis with the minimum S

ChooseSplitIndex

CSI1: Along the chosen axis, chose the distr. with the min overlap-value, then min area-value

R*-Tree - Insert

Insert(R, leaflevel)

I1: Invoke ChooseSubtree with the level as a parameter and find node N

I2: If N has less than M entries, add R in N. If N has M entries, invoke OverflowTreatment(N.level)

I3: If OverflowTreatment was called and a split was performed, propagate upwards. If root split, create new split

I4: Adjust the rectangles accordingly

R*-Tree - Insert

OverflowTreatment(level)

OT1: If level not root.level and this is the first call of OT in level then call ReInsert, else invoke Split

ReInsert

PI1: For all M+1 entries on N, compute the dist between the centers of the entries and the center of N
PI2: Sort the entries in decreasing order of their distance and remove p entries adjusting N's MBR
PI3: Invoke **Insert** to reinsert the entries

R*-Tree - Insert

- R*-tree differs from the R-tree only in the insertion algorithm
- Authors report improvements of up to 50% compared to R-tree
- Also, authors claim that reinsertion helps