Multi-dimensional Indexing

October 21st

Indexing for content based retrieval

- Content based retrieval from large collections of images (more in general of non-textual data) might require support of multi-dimensional index structures to speed-up retrieval.

- We can distinguish two different types of multi-dimensional indexes:

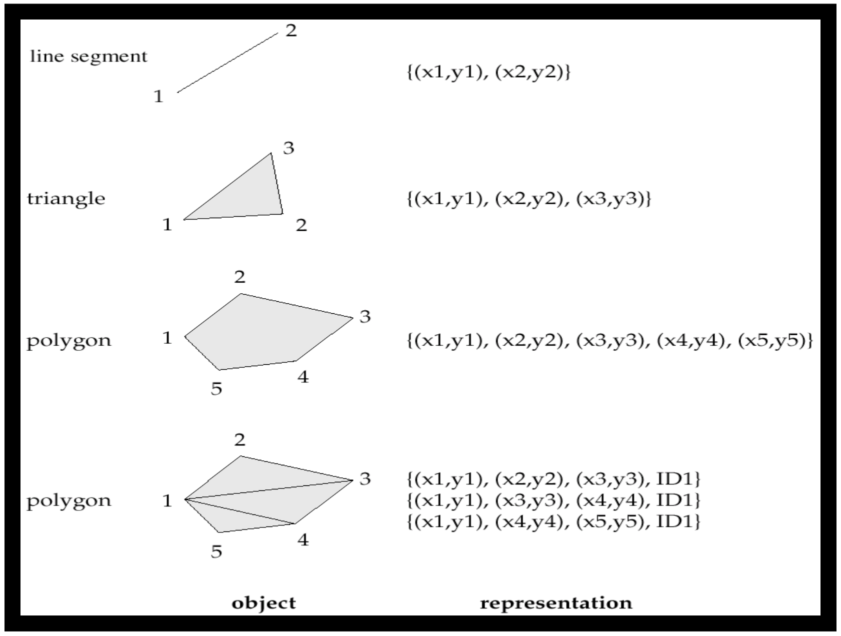
  - Low dimensional indexes, typically used in GIS and spatial database applications. They work with dimensions in the range of 2 - 4.

  - High dimensional indexes, typically used to index high dimensional feature vectors used as descriptors of images or image and video objects. They can work with dimensions in the range of 64 - 500

Low-dimensional Indexing

- Most common multidimensional data with low dimension are spatial data. They represent geometric entities: *points, line segments, triangles*, …. (in 2D) and *polyhedrons* (in 3D).

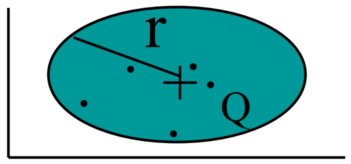
- Geometric entities are usually represented in a normalized fashion:
  - Line segments
    - by the coordinates of their endpoints.
  - Curves
    - by partitioning the curve into a sequence of segments and creating a list of vertices in order
    - …
  - Closed polygons
    - list of vertices in order, starting vertex is the same as the ending vertex,
    - dividing polygon into triangles and note the polygon identifier with each of its triangles.
    - ...


line segment
1
2
{(x1,y1), (x2,y2)}
triangle
1
2
3
{(x1,y1), (x2,y2), (x3,y3)}
polygon
1
2
3
4
5
{(x1,y1), (x2,y2), (x3,y3), (x4,y4), (x5,y5)}
polygon
1
2
3
4
5
{(x1,y1), (x2,y2), (x3,y3), ID1}
{(x1,y1), (x3,y3), (x4,y4), ID1}
{(x1,y1), (x4,y4), (x5,y5), ID1}
object
representation

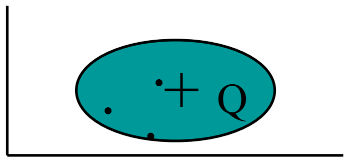
- For applications with spatial data, typical queries are concerned with spatial proximity. Different types of queries are:
  - Range Queries
    - E.g. : *Find all cities within 50 miles of Paris*
    - Query has associated region (location, boundary)
    - Answer includes ovelapping or contained data regions
  - Nearest-Neighbor Queries
    - E.g. : *Find the 10 cities nearest to Paris*
    - Results must be ordered by proximity
  - Spatial Join Queries
    - E.g. : *Find all cities near a lake*
    - Join condition involves regions and proximity

- Most common applications of spatial data are:
  - Geographic Information Systems (GIS)
    - E.g., ArcInfo; OpenGIS Consortium
    - *All classes* of spatial queries and data are common
  - Computer-Aided Design/Manufacturing
    - Store spatial objects such as surface of airplane fuselage
    - *Range queries and spatial join queries* are common

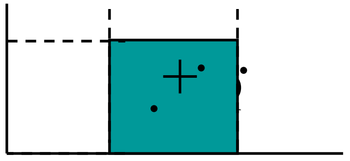
- For applications of content based retrieval in image collections the following query types should be supported:
  - Vague queries: Queries at the earlier stage can be very loose; e.g.: *Retrieve images containing textures similar to this sample.*
  - K-nearest-neighbor-queries: The user specifies the number of close matches to the given query point; e.g. *Retrieve 10 images containing textures directionally similar to this sample*
  - Range queries: An interval is given for each dimension of the feature space and all the records which fall inside this hypercube are retrieved.



r is large
vague query



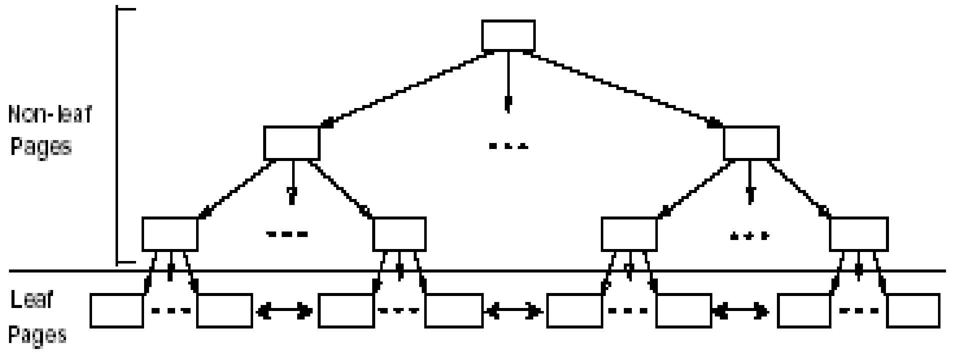
r is small
3-nearest neighbor query



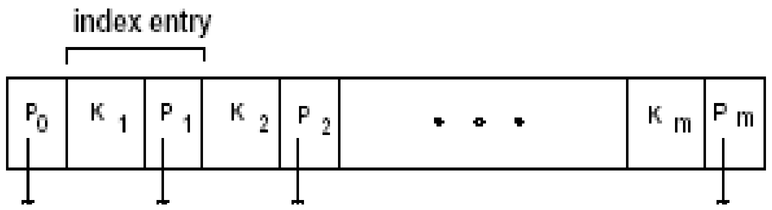
range query

Single-dimensional index

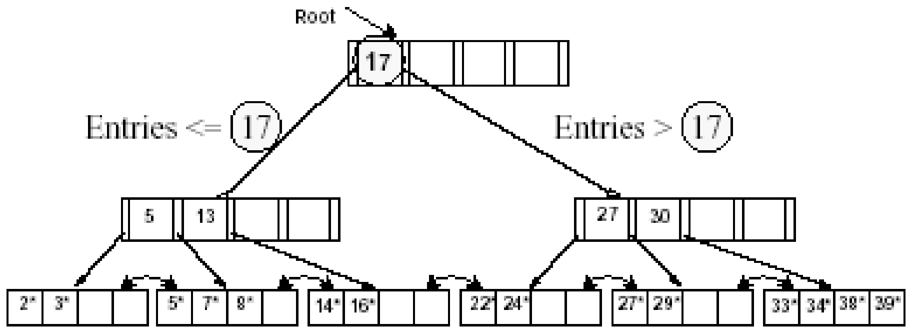
B+ Tree Index



- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages contain *index entries* and direct searches:



Example



- B+ Tree is a dynamic structure for unidimensional data
  - minimum 50% occupancy (except for root);
  - each node contains d <= m <= 2d entries;
  - the parameter d is called the *order* of the tree;
  - typical order: 100;
  - typical fill-factor: 67%.

- B+ Tree supports equality and range-searches efficiently
- Inserts/deletes leave tree height-balanced
- B+ Tree have high fanout (this means depth rarely more than 3 or 4).

- Search cost: O(log (FN)) (F = # entries/index pg, N = # leaf pgs)
  - search begins at root, and key comparisons direct it to a leaf;
  - find 28? 29? : All > 15 and < 30.

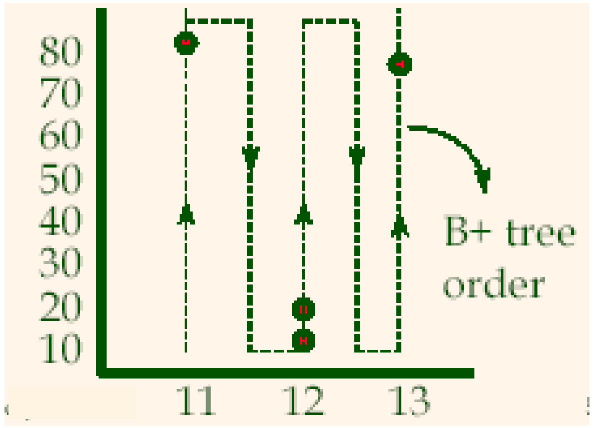
- Insert/delete cost: O(log (FN))
  - find data entry in leaf, then change it;
  - need to adjust parent sometimes;
  - and change sometimes bubbles up the tree.

- B+ Trees can be used for spatial data: when we create a composite search key e.g. an index on <*feature1*, *feature2*>, we effectively linearize the 2-dimensional space since we sort entries first by *feature1* and then by *feature2*.

Consider entries:

<11, 80>, <12, 10>

<12, 20>, <13, 75>



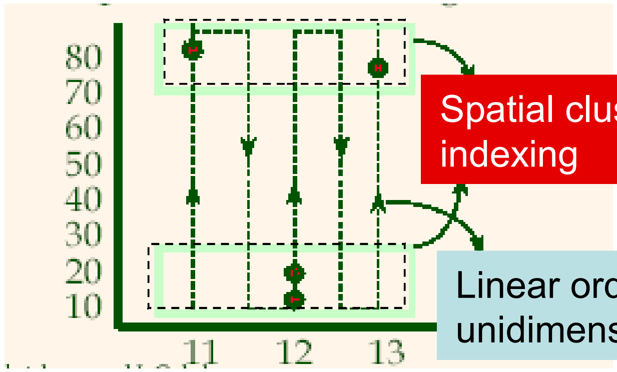
Multi-dimensional indexes

- A multidimensional index clusters entries so as to exploit “nearness” in multidimensional space.
- The basic motivation for multi-dimensional indexing is that for efficient content-based retrieval it is necessary to cluster entities so as to exploit *nearness* in multidimensional space.

.

Consider the entries:

<11, 80>, <12, 10>
<12, 20>, <13, 75>



Spatial clusters with multidimensional indexing

Linear ordering with classical B+tree unidimensional indexing

- Most used multidimensional index structures:
  - k-d Trees
  - Point Quadtrees
  - R, R*, R+ Trees

For low/high-dimensional indexes

  - SS, SR trees
  - M Trees

For very high-dimensional indexes

Low-dimensional indexes

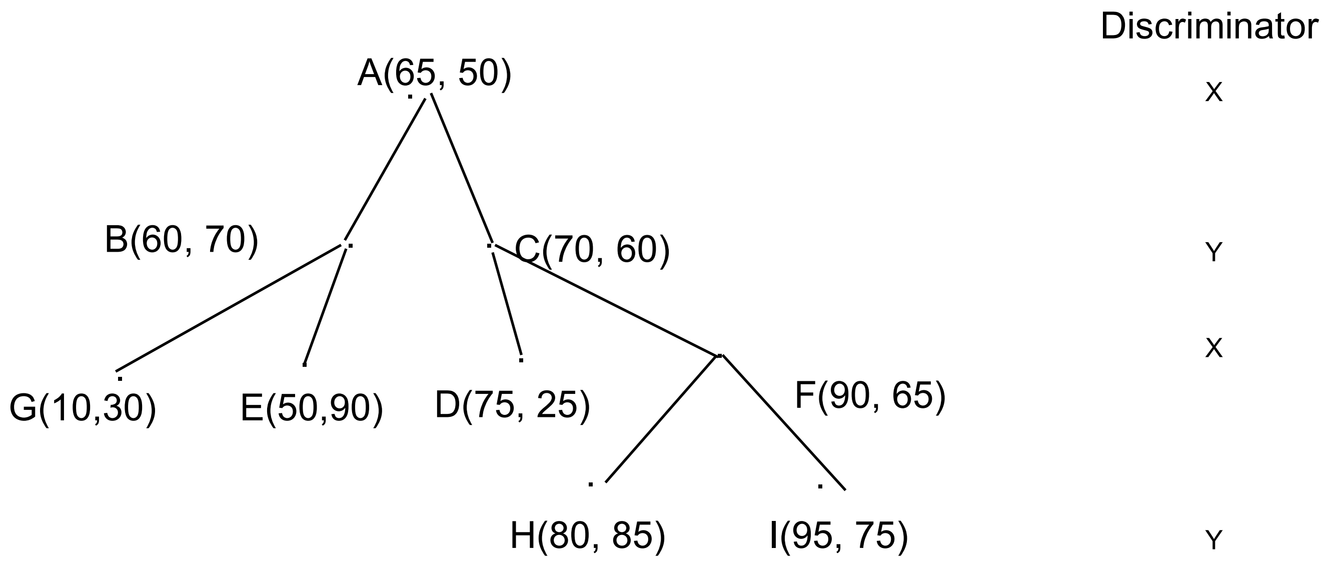
k-d Tree index

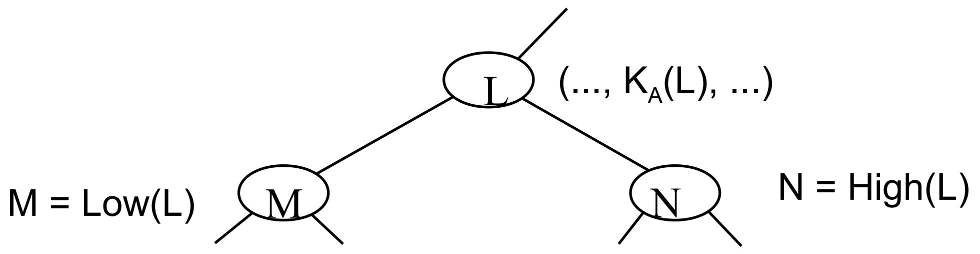
- The k-d Tree index (Bentley 1975) is a multidimensional binary search tree.
- Each node consists of a “record” and two pointers. The pointers are either null or point to another node.
- Nodes have levels and each level of the tree discriminates for one attribute
  - choose one dimension for partitioning at the root level of the tree
  - choose another dimension for partitioning in nodes at the next level and so on cycling through the dimensions.

Example

<u>Input Sequence</u>

A = (65, 50)
B = (60, 70)
C = (70, 60)
D = (75, 25)
E = (50, 90)
F = (90, 65)
G = (10, 30)
H = (80, 85)
I = (95, 75)





Disc(L) : The discriminator at L's level
$K_A(L)$: The A-attribute value of L
Low(L) : The left child of L
High(L) : The right child of L

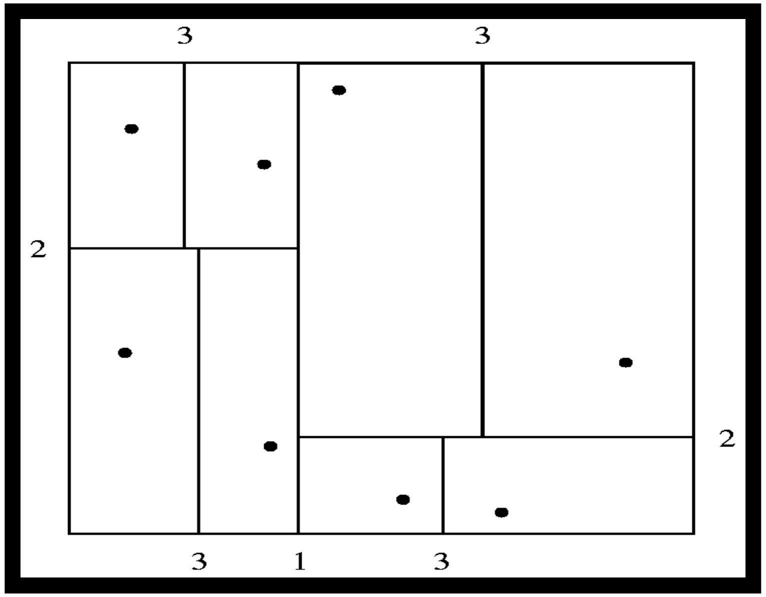
- Search for $P(K_1, ..., K_n)$:

```
Q := Root;
While NOT DONE DO the following:
    if $K_i(P) = K_i(Q)$ for i = 1, ..., n then we have located the node and we are DONE
    Otherwise
        if A = Disc(Q) and $K_A(P) < K_A(Q)$ then Q := Low(Q)
        else Q := High(Q)
```

- Performance: O(logN), where N is the number of records

2-d Tree index

- 2-d Trees can be used for indexing point data in spatial databases.
- Division of space with 2-d Trees:
  - each line in the figure corresponds to a node in the k-d tree
  - In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
  - Partitioning stops when a node has less than a given maximum number of points.



The numbering of the lines indicates the level of the tree at which the node appears.
The maximum number of points in a leaf node is set to 1.

- 2-d Trees are effective index structures for *range queries*.

  Example of range query: *search those nodes of coordinates xy whose distance from vector (a,b) is less than d*
  - The minimum cannot be less than a-d, b-d
  - The maximum cannot be greater than a+d, b+d

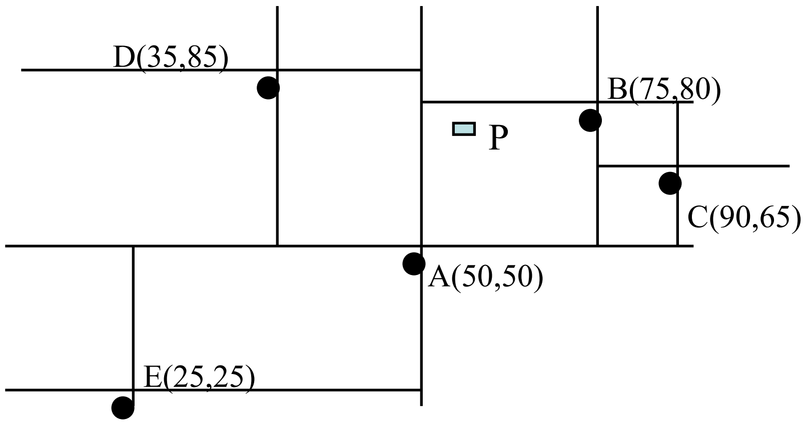
  Therefore at each node x,y values must be compared with values stored in the node to decide which subtree has to be discarded

k-d Quadtree index

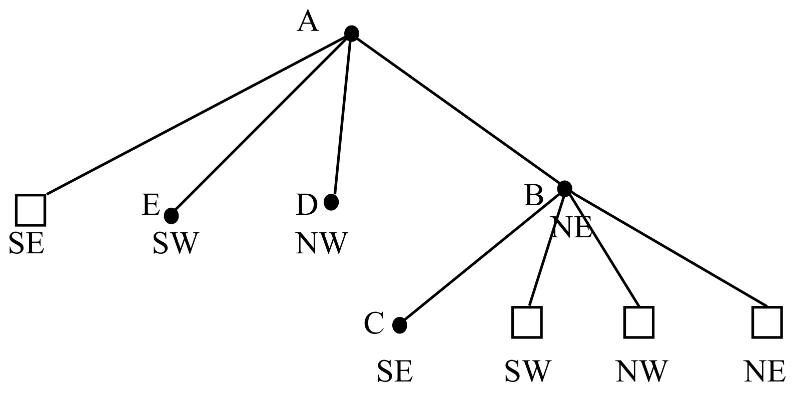
- In a k-dimensional Quadtree each node partitions the object space into quadrants.The partitioning is performed along all search dimensions and is data dependent, like k-d Trees.

Example of Quadtree for 2D points:

Partitioning of the space



The Quadtree



To insert P(55, 75):

- Since $X_A < X_P$ and $Y_A < Y_P$ go to NE (i.e., B).
- Since $X_B > X_P$ and $Y_B > Y_P$ go to SW, which in this case is null.

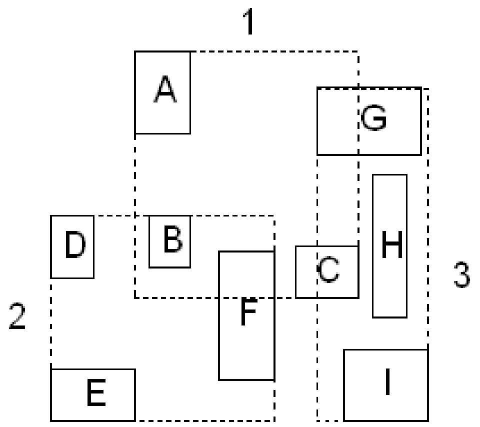
High-dimensional Indexes

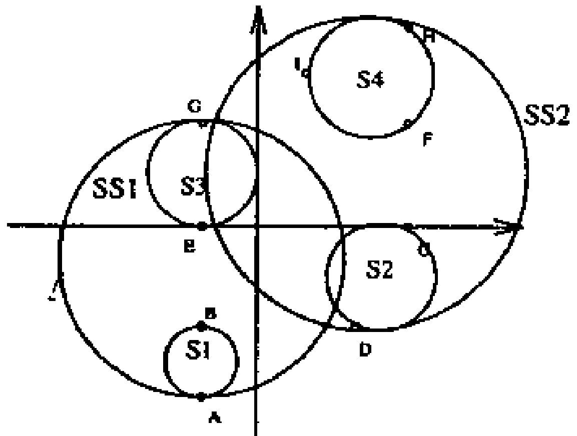
- Image data are in two forms:
  - n-dimensional points (a feature vector describing the image or image objects)
  - n-dimensional objects with some spatial extension (for the evaluation of spatial relations)

- For image data high/very high dimensions indexes are needed. Typically these indexes should perform partitioning of the embedding space according to Minimum Bounding Regions (MBR) or Minimum Bounding Spheres (MBS) in the n-dimensional space. MBR and MBS refer to the smallest rectangle or circle respectively that encloses the n-dimensional entity (a region or a feature vector in n-d).

- As the number of dimensions increases the performance tends to degrade and most indexing structures become inefficient for certain types of queries

In 2D

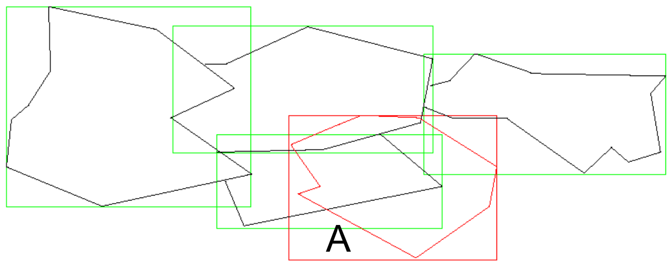




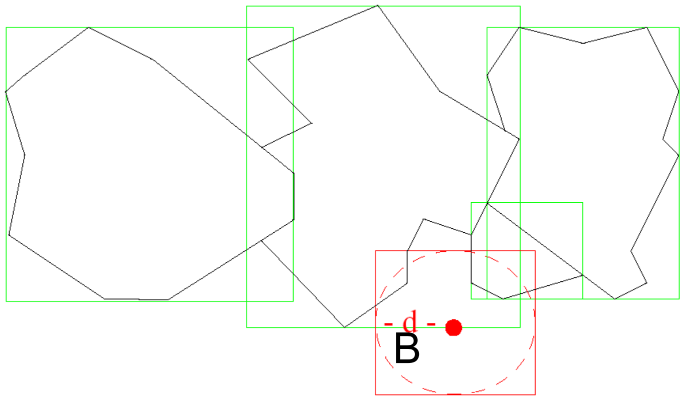
- Rectangles and circles are more difficult than points since do not fall into a single cell of a bucket partition. Therefore with these index structures it is necessary to manage overlapping of bucket regions.

- These index structures offer support also for new types queries:

  - Adjacency query: find regions adjacent to *A*.

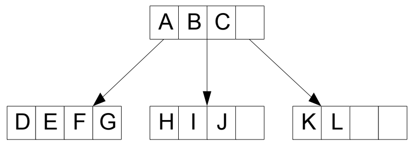


  - Distance qualified query: find regions within distance *d* from *B*.

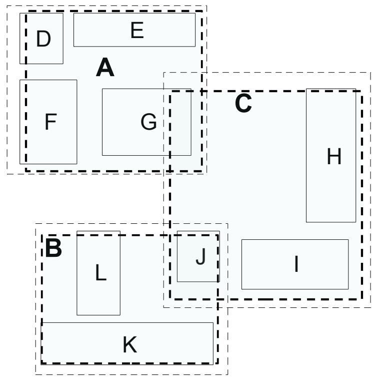


R Tree index

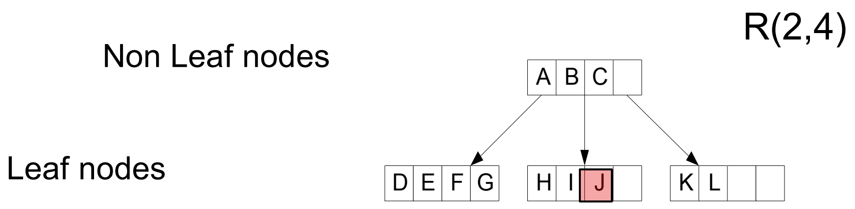
- The R Tree (Guttmann 1984) is a tree structured index that remains balanced on inserts and deletes. R Trees have ben designed for indexing sets of rectangles and other polygons. Each key stored in a leaf entry is intuitively a box, or *collection of intervals*, with one interval per dimension.
- R Trees partition the space into rectangles, without requiring the rectangles to be disjoint. Enclosing rectangular regions are drawn with sides coinciding with the sides of the enclosed rectangles
- R Tree is supported in many modern database systems, along with variants like R^+ Trees and R* Trees.

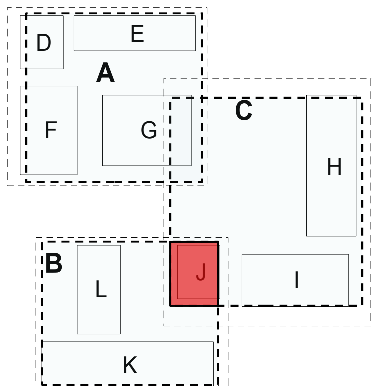


Example in 2D:



- Root and intermediate nodes correspond to the smallest rectangle that encloses its child nodes. Leaf nodes contain pointers to the actual objects:
  - Non-leaf entry = < *n-dim box, ptr to child node* > (box covers all boxes in child node (subtree))
  - Leaf entry = < *n-dimensional box, rid* > (box is the tightest bounding box for a data object)
- All leaf nodes appear at the same level (same distance from root).
- A rectangle may be spatially contained in several nodes (see rectangle J in the example), yet it can be associated with only one node.





- The R Tree order (n,M) is the minimum and maximum number of rectangles stored at each node.
- An R Tree of order (n,M) contains at each node between n <= M/2 and M entries

- Search for objects overlapping box Q
  Start at root.
  - If current node is non-leaf, for each entry <E, ptr>,
    - if box E overlaps Q, search subtree identified by ptr.
  - If current node is leaf, for each entry <E, rid>,
    - if E overlaps Q, rid identifies an object that might overlap Q.

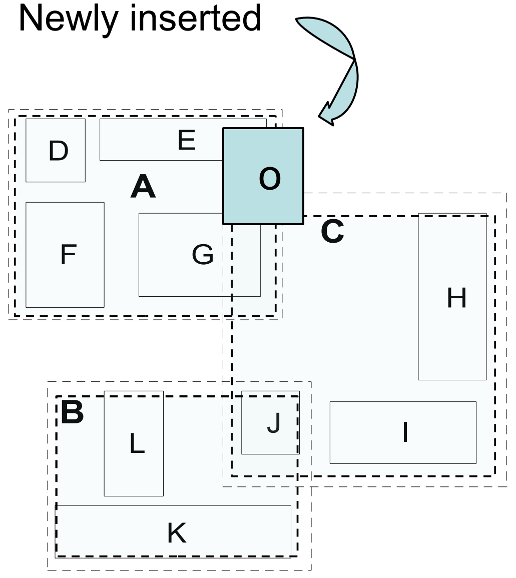
  - May have to search several subtrees at each node

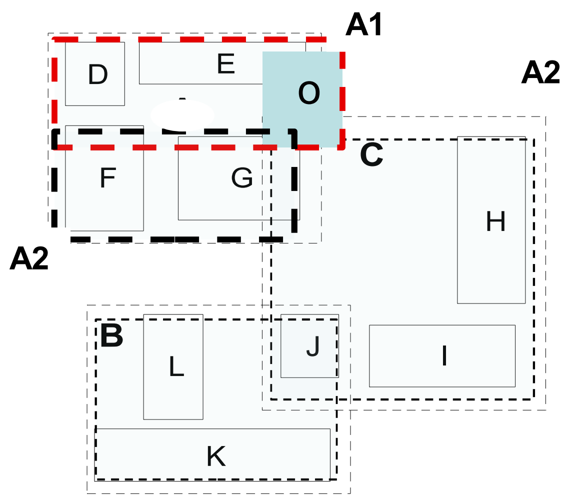
- Delete
  - Delete consists of searching for the entry to be deleted, removing it, and if the node becomes under-full, deleting the node and then re-inserting the remaining entries.
  - Remaining nodes are not merged with adjacent nodes as in B Tree.
    There is no concept of adjacency in an R Tree.

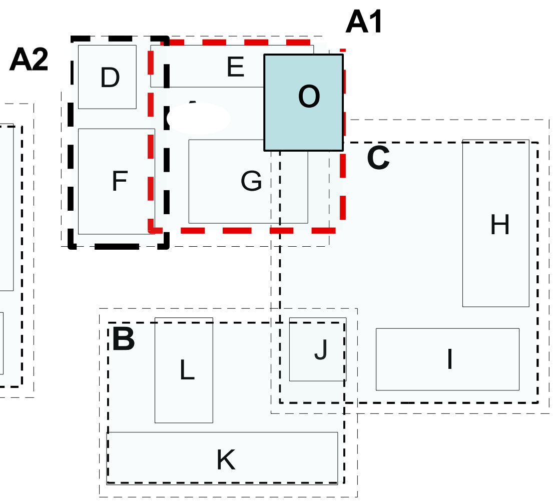
- Insertion
  - A new object is added to the appropriate leaf node.
  - If insertion causes the leaf node to overflow, the node must be split, and the records distributed in the two leaf nodes. Goal is to reduce likelihood of both L1 and L2 being searched on subsequent queries:
    1. Minimizing the total area of the covering rectangles
    2. Minimizing the area common to the covering rectangles
  - Splits are propagated up the tree (similar to B tree).

- Start at root and go down to best fit leaf L.
- Go to child whose box needs least enlargement to cover O
- Resolve ties by going to smallest area child
- If best fit leaf L has space, (A in ex) insert entry and stop. Otherwise, split L into L1 and L2 (A1, A2 in ex)
- Adjust entry for L in its parent so that the box now covers only L1.
- Add an entry in the parent node of L for L2. This could cause the parent node to recursively split.

Newly insertedGood splitBad split







R(2,4)

Considerations on R Trees

- For search
  - advantage: each spatial object (or key) is in a single bucket
  - disadvantage: multiple search paths due to overlapping bucket regions.
  - can improve search performance by using a convex polygon to approximate query shape (instead of a bounding box) and testing for polygon-box intersection.

- Generalization for higher dimension is straightforward, although R Trees work well only for relatively small n

R* Tree index

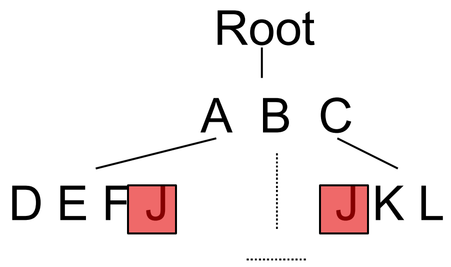
- The R* Tree is a variant of R Tree that uses the concept of forced reinserts to reduce overlap in tree nodes. When a node overflows, instead of splitting:
  - Remove some (say, 30% of the) entries and reinsert them into the tree.
  - Could result in all reinserted entries fitting on some existing pages, avoiding a split.

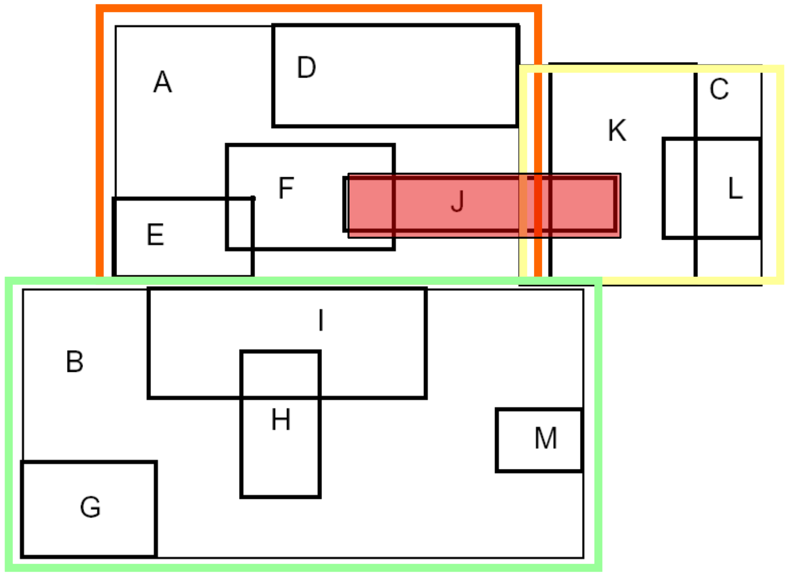
- R* Trees also use a different heuristic, minimizing box perimeters rather than box areas during insertion.

R+ Tree index

- R+ trees (Sellis, Rossopoulos & Faloutsos 87) are an alternative to R Trees that avoids overlap of enclosing rectangles by inserting an object into multiple leaves if necessary. One rectangle is associated with all the enclosing rectangles that it intersects.

- Data rectangles cut into several pieces, if necessary. The same rectangle can be reached from multiple paths starting from the root. Searches now take a single path to a leaf, at cost of redundancy.





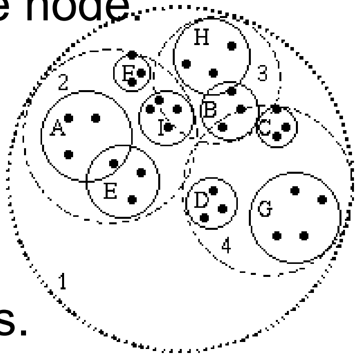
Very High-dimensional indexes

- Typically, high-dimensional datasets are collections of points, not regions and are very sparse.
  - E.g. feature vectors in multimedia applications.
  - R-tree becomes worse than sequential scan for most datasets with more than a dozen dimensions.
- As dimensionality increases, contrast (ratio of distances between nearest and farthest points) usually decreases;
  - “nearest neighbor” is not any more meaningful.
  - In any given data set, empirically test contrast.

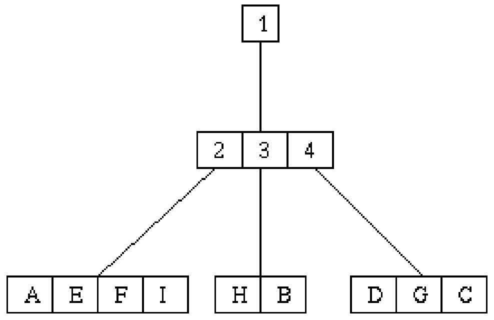
SS Tree index

- The SS Tree index uses minimum bounding spheres (MBSs) to represent objects instead of MBRs. SS Tree index divides points into short-diameter regions, while R Tree divides points into small-volume regions.

- While 2 x d real numbers are used to represent MBRs, only d+1 real numbers are used for MBSs (one for the sphere center and one for its radius).The space saving allows more entries to be fit in a tree node.



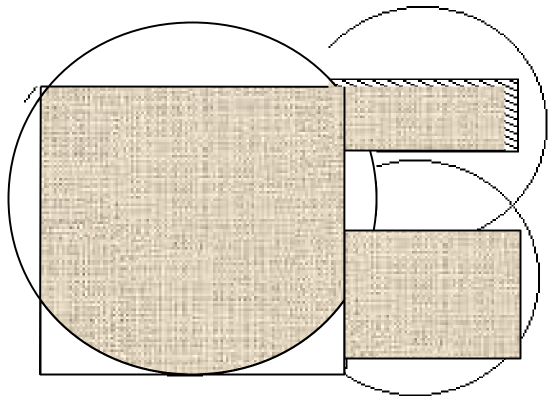
- In the SS Tree, points are divided into isotropic neighborhoods. The center of a sphere is the centroid of underlying points.



SR Tree index

- SR Tree index are based on the consideration that minimum bounding spheres occupy much larger volume than bounding rectangles with high dimensional data on average, and this reduces search efficiency.

- According to this in the SR Tree index, a region is specified by the intersection of a bounding sphere and a bounding rectangle.



- The SR Tree index combines the advantages of both spherical and rectangular regions. In fact:
  - the average volume of bounding rectangles is much smaller than that of bounding spheres;
  - the average diameter of bounding rectangles is much longer than that of bounding spheres.

Distance Based Index Structures

- Distance based index structures also referred to as *metric trees* are proposed for applications where:
  - the distance computation between objects of the data domain are expensive (such as for high dimensional data)
  - the distance function is metric

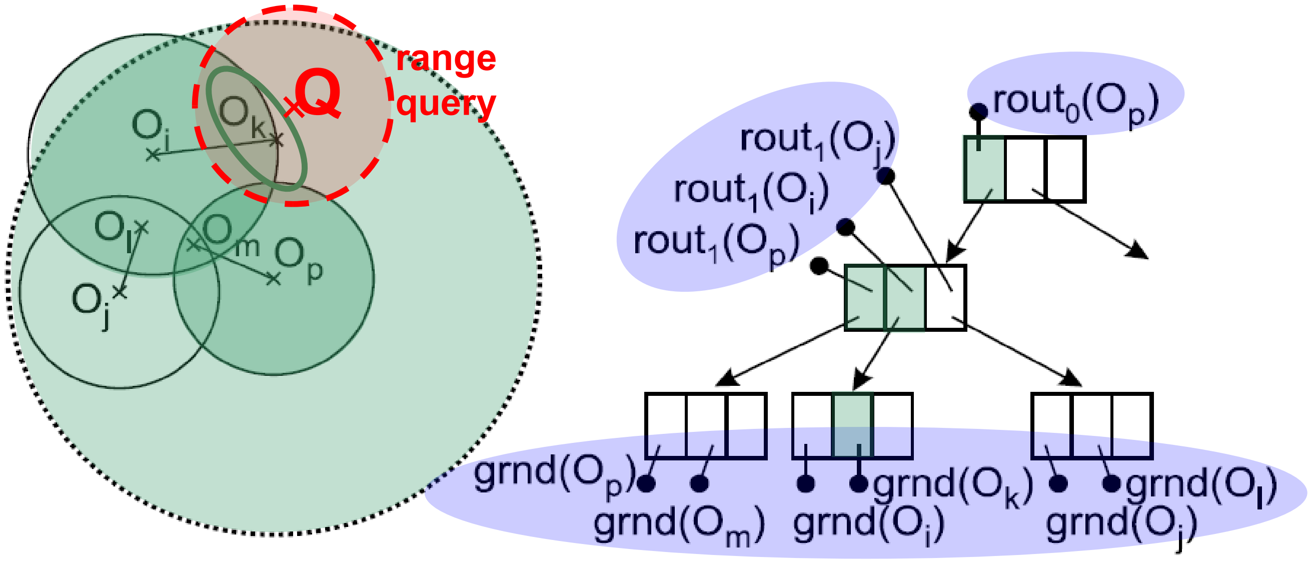
- Metric trees only consider relative distances of objects to organize and partition the search space. The only requirement is that the distance function is metric, which allows the triangle inequality to be applied.

- Since metric spaces strictly include vector spaces, metric trees have more general applicability in content-based retrieval than multidimensional access methods.

M Tree index

- M Tree is a balanced paged metric tree like e.g. B^+ Tree and R Tree, designed to act as a dynamic access method. Objects are collected in a hierarchical set of clusters:
  - The tree leaves are clusters of indexed objects O_i (ground objects)
  - Routing entries in the inner nodes represent hyper-spherical metric regions (O_i , r_{Oi}), recursively bounding the object clusters in leaves
  - Each cluster has:
    - A reference (routing) object;
    - A radius providing an upper bound for the maximum distance between the reference object and any other object in the cluster.

- The compactness of metric regions' hierarchy in M-tree heavily depends on the order of new objects' insertions



(Euclidean 2D space)

- The triangle inequality allows to discard irrelevant M-tree branches (metric regions resp.) during query evaluation

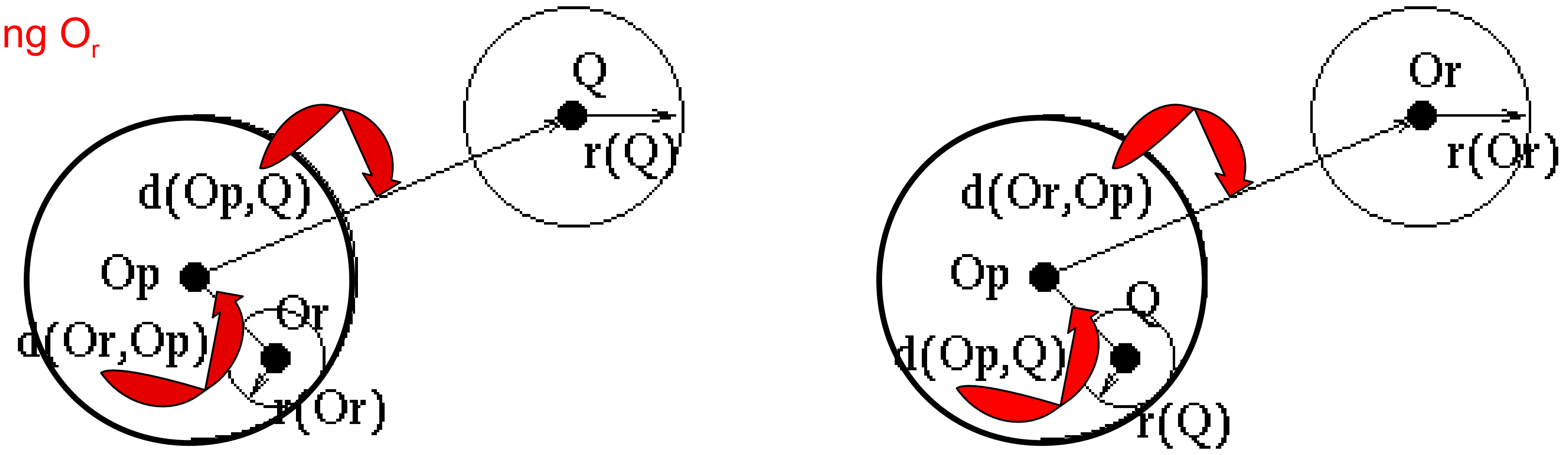
- To this end, an entry of the M Tree index in an internal node is constructed as follows:

$$entry(O_r) = \left[O_r, ptr(T(O_r)), r(O_r), d(O_r, P(O_r))\right]$$

- where:
  - O_r is the *routing object*
  - *ptr(T(*O_r*))* is the pointer to the root of sub-tree *T(*O_r*)* - the *covering tree* of O_r
  - *r(*O_r*)* is the *covering radius* of O_r
  - *d(*O_r*,P(*O_r*))* is the distance from the parent object

- Since the distance is metric, during search, the triangular inequality is used to prune clusters that are bound out of an assigned range from the query.

Pruning O_r

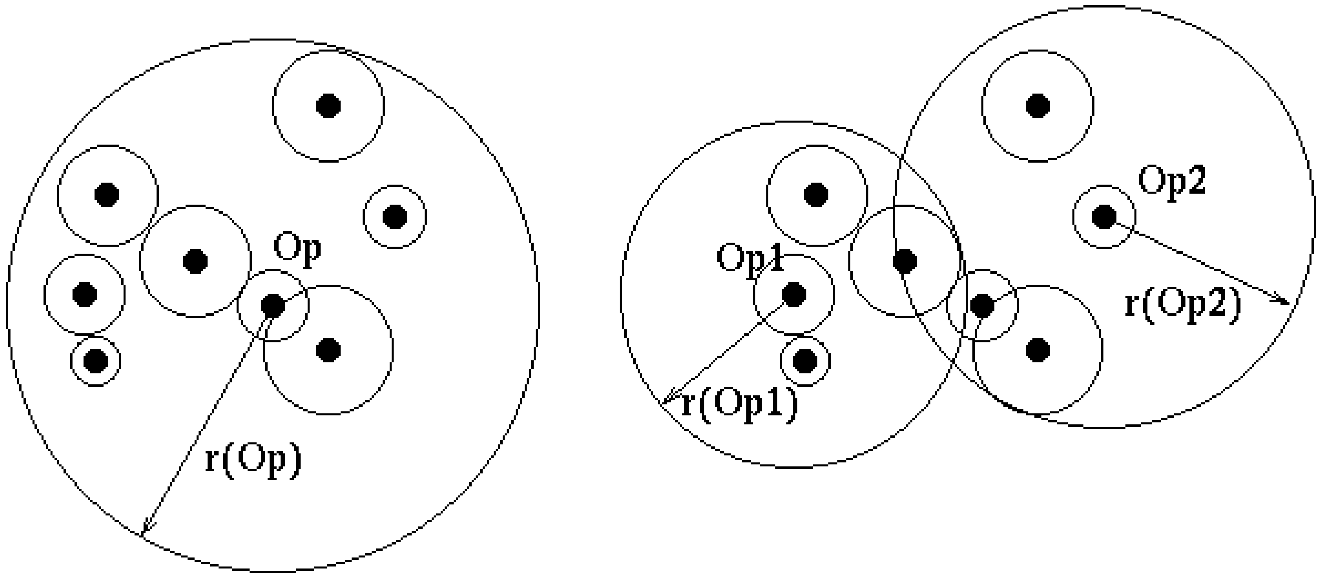


Given the range query (Q,r(Q)), the condition applied for pruning are as follows:

T(O_r) can be safely pruned from the search if: $d(O_r, Q) > r(Q) + r(O_r)$

or if: $|d(O_p,Q) - d(O_r,O_p)| > r(Q) + r(O_r) \Rightarrow d(O_r,Q) > r(Q) + r(O_r)$

- M-tree splitting


Op
r(Op)
Op1
r(Op1)
Op2
r(Op2)