15 Indexing Spatial Data

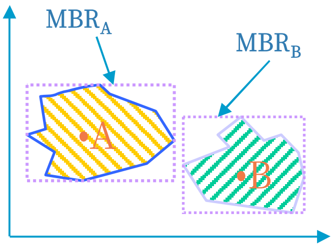
Introduction

Indexing Spatial Data 15

- Many applications(e.g., CAD, GIS) operate on *spatial data*, which include points, lines, polygons and so on
- Conventional DBMSs are unable to support spatial data processing efficiently
  - First, spatial data are large in quantity, complex in structures and relationships
  - Second, the retrieval process employs *complex spatial operators* like *intersection*, *adjacency*, and *containment*
  - Third, it is difficult to define a spatial ordering, so conventional techniques(e.g., sort-merge) cannot be employed for spatial operations
- Spatial indexes need!

Query Processing

- It is expensive to perform spatial operations (e.g., intersect, contain) on real spatial data
- Thus, simpler structure that *approximates* the objects are used: Minimum Bounding Rectangle or circle
- Example: intersection



- **Step1**: perform intersection operation between MBR_A and MBR_B
- **Step2**: if MBR_A intersects with MBR_B, then perform intersection operation between A and B

Query Processing (Con't)

- Multi-step spatial query processing
  1. The spatial index prunes the search space to a set of candidates
  2. Based on the approximations of candidates, some of the *false hit*s can be further filtered away
  3. Finally, the actual objects are examined to identify those that match the query
  - The multi-step strategy can effectively reduce the number of pages accessed, the number of data to be fetched and tested and the computation time through the approximations
  - Types of spatial queries
    - Spatial selection: point query, range(window) query
    - Spatial join between two or more different entities sets

A Taxonomy of spatial indexes

▶ Classification of spatial indexes

1. The transformation approach
   - Parameter space indexing
     - Objects with *n* vertices in a *k*-dimensional space are mapped into points in a *nk*-dimensional space
     - e.g.) two-dimensional rectangle described by the two corner (x_1,y_1) and (x_2, y_2) => a point in a four-dimensional space
   - Mapping to single attribute space
     - The data space is partitioned into grid cells of the same size, which are then numbered according to some *curve-filling methods*(e.g., hilbert curve, z-ordering, snake-curve)

A Taxonomy of spatial indexes (Con't)

▶ Classification of spatial indexes

2. The non-overlapping native space indexing approach
   - Object duplication
   - Object clipping

A Taxonomy of spatial indexes (Con't)

- Classification of spatial indexes
  3. The overlapping native space indexing approach
     - Partitioning hierarchically the data space into a manageable number of smaller subspaces
     - Allowing the overlapping between bounding subspaces
     - The overlapping minimization is very important
     - e.g.)
       – binary-tree: kd-tree, LSD-tree, etc.
       – B-tree: k-d-b-tree, R-tree, R*-tree, TV-tree, X-tree, etc.
       – Hashing: Grid-files, BANG files, etc.
       – Space-Filling: Z-ordering, Filter-tree, etc.

Binary-tree based indexing

- The characteristics
  – A basic data structure for representing data items whose index values are ordered by some linear order
  – Repetitively partitioning a data space
- Types of binary search trees
  – kd-tree
  – K-D-B-tree
  – hB-tree
  – skd-tree
  – LSD-tree

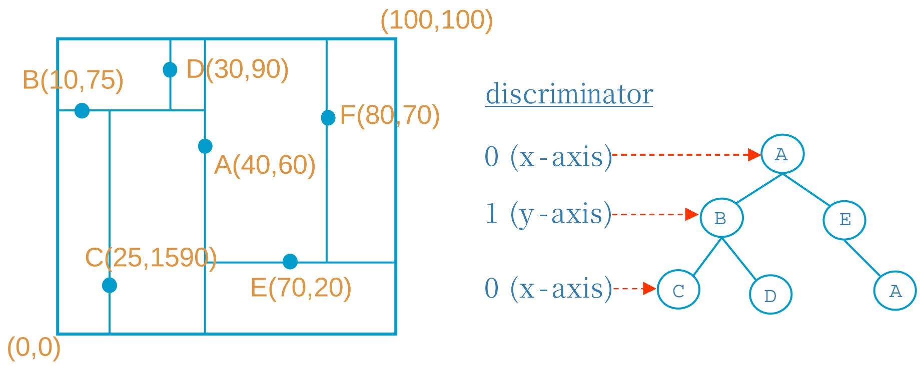
Binary-tree based indexing: The *kd*-tree

- The kd-tree
  - k-dimensional binary search tree to index multi-attribute data
  - A *node* in the tree serves both representation of a actual data point and direction of search
  - A *discriminator* is used to indicate the key on which branching decision depends
  - A node P has two children, a left son LOSON(P) and a right son HISON(P)
  - If discriminator is the j^{th} attribute, then the j^{th} attribute of any node in the LOSON(P) is less than the j^{th} attribute of node P, and the j^{th} attribute of any node in the HISON(P) is greater than or equal to that of node P

Binary-tree based indexing: The *kd*-tree (con't)

- The kd-tree
  - Complications arise when an internal node(Q) is deleted
    - One of the nodes in the subtree whose root is Q must replace Q
    - To reduce the cost of deletion, a non-homogeneous kd-tree was proposed
  - The kd-tree has been the subject of intensive research over the past decade: clustering, searching, storage efficiency and balancing

Binary-tree based indexing: The *kd*-tree (con't)



(a) data space

(b) *kd*-tree

Binary-tree based indexing: The *K-D-B*-tree

- The K-D-B-tree
  - is a combination of a *kd*-tree and B-tree
  - consists of a *region page* and a *point page*
    - region page: <region, page-ID> pairs
    - point page: <point, record-ID> pairs
  - is perfectly height-balanced
  - has poorer storage efficiency, nevertheless

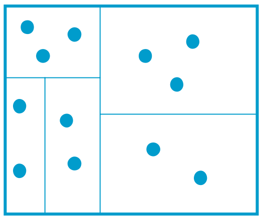
Binary-tree based indexing: The *K-D-B*-tree (Con't)

- Splitting
  - data page splitting
    - A split will occur during insertion of a new point into a *full* point page
    - The two resultant point pages will contain almost the same number of data points
    - The split of a point page may cause the parent region page to split as well, which may propagate upward

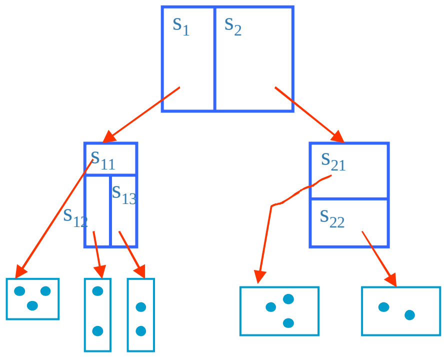
Binary-tree based indexing: The *K-D-B*-tree (Con't)

- Splitting
  - region page splitting
    - A split will occur when a region page is full
    - A region page is partitioned into two region pages such that both have almost the same number of entries
    - The split may propagate downward
    - The downward propagation may cause low storage utilization

Binary-tree based indexing: The *K-D-B*-tree (Con’t)



(a) k-space

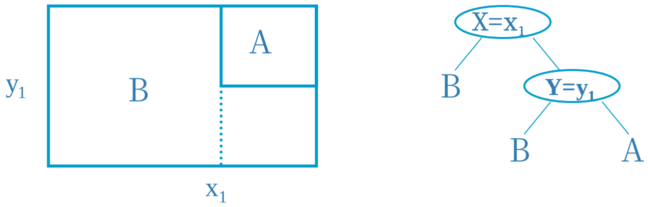


(b) K-D-B Tree

Binary-tree based indexing: The *hB*-tree

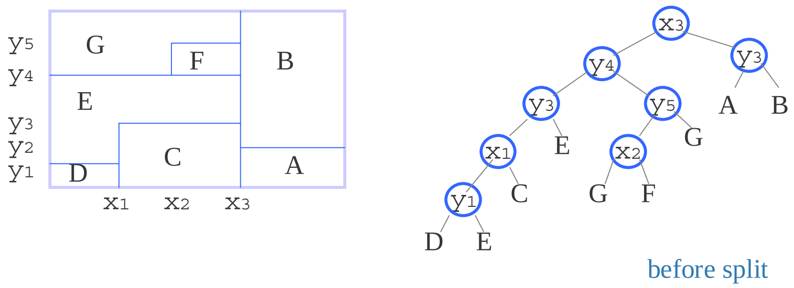
- The hB-tree
  - problem in the K-D-B-tree
    - The split of one index node can cause descendant nodes to be split as well. This may cause sparse index nodes to be created
  - To overcome this problem, the hB-tree (the holey brick B-tree) allows the data space to be holey
  - Based on the K-D-B-tree => height-balanced tree
  - Data nodes + Index nodes
  - Data space may be non-rectangular and kd-tree is used to space representation in internal nodes

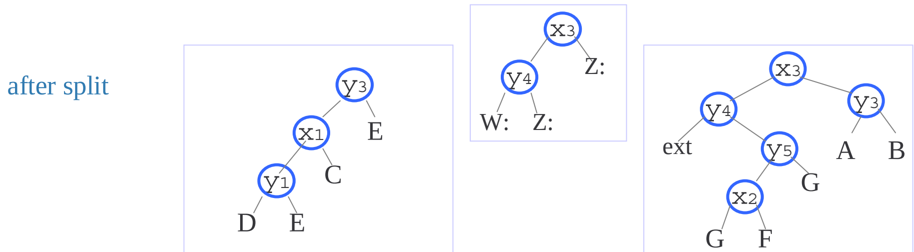
Binary-tree based indexing: The *hB*-tree (Con’t)



A holey brick is represented via a kd-tree. A holey brick is a brick from which a smaller brick hash been removed. Two leaves of the kd-tree are required to reference the holey brick region denoted by B.

Binary-tree based indexing: The *hB*-tree (Con’t)





Binary-tree based indexing: The *hB*-tree (Con't)

- The advantages
  - Overcoming the problem of sparse nodes in the K-D-B-tree
  - The search time and the storage space are reduced because of the use of kd-tree
- The disadvantages
  - The cost of node splitting and node deletion are expensive
  - The multiple references to data nodes may cause a path to be traversed more than once

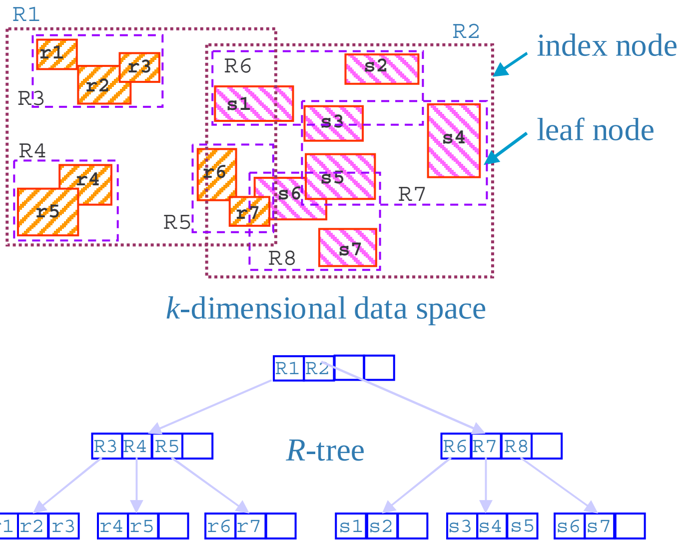
B-tree based indexing: The R-tree

- The R-tree
  - A multi-dimensional generalization of the B-tree
  - A height-balanced tree
  - Having received a great deal of attention due to its well defined structure
  - Like the B-tree, node splitting and merging are required

B-tree based indexing: The R-tree (Con't)

- The structure of the R-tree
  - Leaf node : a set of entries of <MBR, *object-ID*>
    - MBR: a bounding rectangle which bounds its data object
    - object-ID: an object identifier of the data object
  - Index node : a set of entries of <MBR, *child-pointer*>
    - MBR: a bounding rectangle which covers all MBRs in the lower node in the subtree
    - child-pointer: a pointer to a lower level node in the subtree

B-tree based indexing: The R-tree (Con't)



k-dimensional data space

R-tree

B-tree based indexing: The R-tree (Con't)

- Search
  - Query operations: intersect, contain, within, distance, etc.
  - Query rectangle: a rectangle represented by user
  - The search algorithm
    - Recursively traverse down the subtrees of MBR which intersect the query rectangle
    - When a leaf node is reached, MBRs are tested against the query rectangle and then their objects are tested if they insect the query rectangle
  - Primary performance factor: minimization of overlaps between MBRs of index nodes => determined by the splitting algorithm(different from other R-tree variants)

B-tree based indexing: The R-tree (Con't)

- Insertion
  - Criterion: least coverage
    - The rectangle that needs *least enlargement* to enclose the new object is selected, the one with the *smallest area* is chosen if more than on rectangle meets the first criterion
- Deletion
  - In case that the deletion causes the leaf node to underflow, the node is deleted and all the remaining entries of that node are reinserted from the root

B-tree based indexing: The R*-tree

▶ The R*-tree

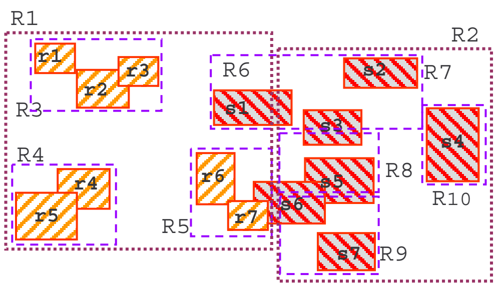
- Minimization of both coverage and overlap is crucial to the performance of the R-tree. So the near optimal of both minimization was introduced by Beckmann et al.: The criterion that ensures the *quadratic* covering rectangles is used in the insertion and splitting algorithms
- Dynamic hierarchical spatial indexes are sensitive to the order of the insertion of data: Beckmann proposed a *forced reinsertion* algorithm when a node overflows`

B-tree based indexing: The R^+-tree

▶ The R^+-tree

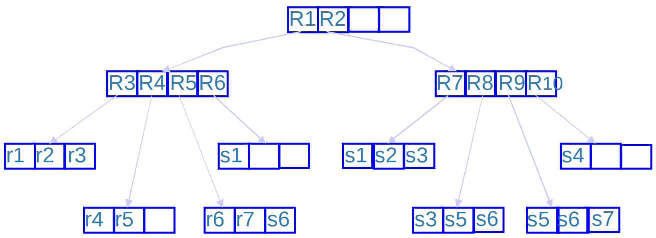
- A compromise between the R-tree and the K-D-B-tree
- Overcoming the problem of the overlapping of internal nodes of the R-tree
- The R^+-tree differs from the R-tree:
  - Nodes of an R^+-tree are no guaranteed to be at least half filled
  - The entries of any internal node do not overlap
  - An object identifier may be stored in more than one leaf node
- The disjoint MBRs avoid the multiple search paths for point queries

B-tree based indexing: The R^+-tree (Con’t)



k-dimensional data space

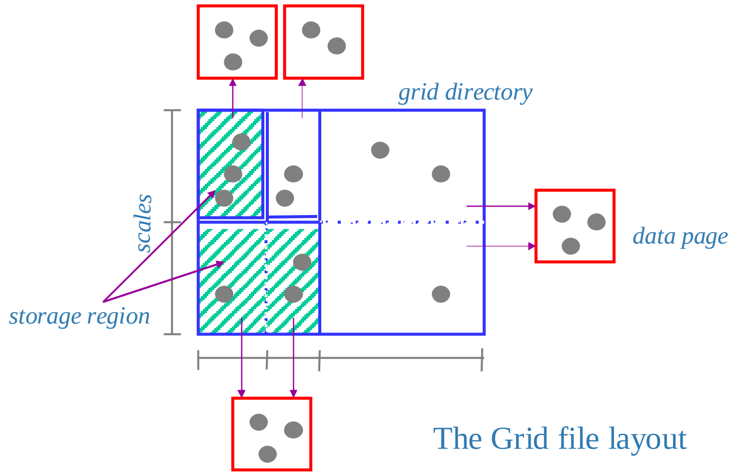
R^+-tree



Cell methods based on dynamic hashing: The grid file

- The grid file
  - Based on dynamic hashing for multi-attribute point data
  - Two basic structures: *k linear scales* + *k-dimensional directory*
  - *grid directory*: k-dimensional array
  - Each grid need not contain at least *m* objects. So a data page is allowed to store objects from several grid cells as long as the union of these cells from a rectangular rectangle, which is known as the *storage region*

Cell methods based on dynamic hashing: The grid file (Con’t)

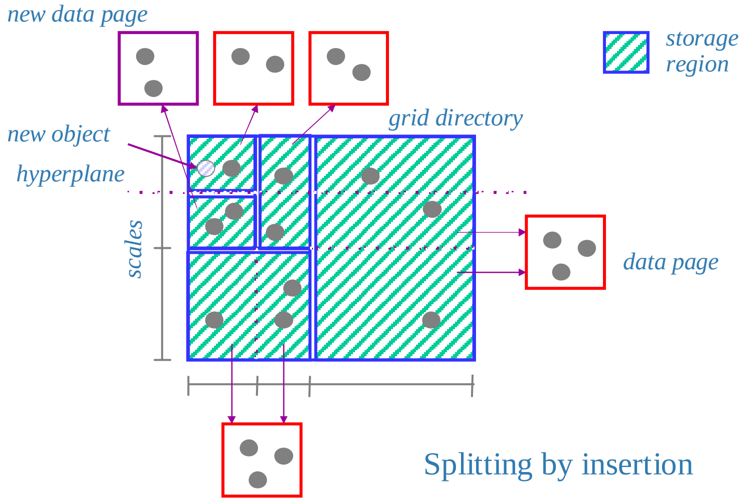


The Grid file layout

Cell methods based on dynamic hashing: The grid file (Con’t)

- ▶ Splitting by insertion
  - – In the case where the data page is *full*, a split is required
    - The split is simple if the storage region covers more than the grid cells
    - Otherwise a new (*k*-1)-dimensional hyperplane partitions the corresponding storage region into two subspaces
      - – The corresponding storage region: partition into two regions and distribute objects into the existing page and a new data page
      - – Other storage regions: unaffected

Cell methods based on dynamic hashing: The grid file (Con't)



Splitting by insertion

Cell methods based on dynamic hashing: The grid file (Con't)

- Merging by deletion
  - Deletion may cause the occupancy of a storage region to fall below an acceptable level, which triggers merging operations
  - If the joint occupancy of two or more adjacent storage regions drops below a threshold, then the data pages are merged into one
  - Two merging approaches: *neighbor* system and *buddy* system

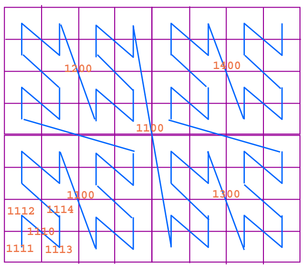
Spatial objects ordering

- ▶ The space-filling curves
  - – Mapping multi-dimensional objects to one-dimensional values
    - • Numbering each grid in a space according to mapping function (e.g., Peano-Hilbert curve, z-ordering, gray-ordering, etc.)
    - • one-dimensional locational key is a number
  - – A B^+-tree is used to index the objects based on locational keys

Indexing Spatial Data 15

Spatial objects ordering (Con’t)

e.g.) z-ordering



z-ordering

$$k = \begin{cases} k' + 5^{m-h} & \text{if k is the SW son of k'} \\ k' + 2*5^{m-h} & \text{if k is the NW son of k'} \\ k' + 3*5^{m-h} & \text{if k is the SE son of k'} \\ k' + 4*5^{m-h} & \text{if k is the NE son of k'} \end{cases}$$

mapping function

Indexing Spatial Data 15