5.1 Quadtree
5.2 R-tree
5.3 K-d tree
5.4 BSP tree
5.5 Grid file
5.6 Summary
5 Spatial Access Methods

• Speeding up queries by well-directed access on relevant tuples

  — Query types
  • Point query
  • Range query
  • Nearest neighbour query

  — Problems
  • Maintenance of the topological structure
  • Density of objects varies strongly
  • Dynamic reorganisation
  • Representation of objects: points as well as extended objects

http://www.forst-ib-espig.de/
For one-dimensional data: B-Tree

- Upper (U) and lower (L) bound for number of links usually: \(2L = U\)
- Insertion
  - All insertions happen at the leaf nodes
  - Nodes are split during insertion as soon as they contain more than \(U-2\) keys
  - The median is chosen from among the leaf's elements and the new element
  - The splitting may go all the way up to the root
- Deletion
  - Rebalancing necessary if node contains less than \(L\) keys
5 Spatial Access Methods

- Very efficient data structure for disk storage
  - $O(\log n)$ for all operations
  - Even better
    - Guaranteed maximum node-accesses to locate a key is $\left\lceil \log_{\text{fan-out}} \left( \frac{n + 1}{2} \right) \right\rceil$
    - Balanced binary tree guarantees only $\lceil \log_2 n \rceil$
    - Accessing a node is expensive on disks $\rightarrow$ huge improvement

- No degenerated cases
  - Self-balancing rarely necessary as most updates affect just one node
  - Wasted space decreased due to guaranteed minimal fill factor
5 Spatial Access Methods

• **B⁺-Tree**
  – Different nodes for leaf nodes and internal nodes
    • Data pointers only in leaf nodes
  – All leafs are linked to each other in-order
  – Advantages
    • Improved traversal performance
    • Increased search efficiency
    • Increased memory efficiency
5 Spatial Access Methods

• One-dimensional orderings
  – Mapping multidimensional data to one dimension
  – Try to preserve object proximity
  – Use a uniform grid to partition the space
  – Assign a unique number to each cell in a way that
    • Adjacent cells get similar numbers
    • It can be defined by a nonterminating recurrence
    • Each cell can be refined separately without disturbing the order of the other numbers
  → Space filling curve
Examples for space filling curves:

- Z-order
- Gray code
- Hilbert’s curve

http://dbs.uni-leipzig.de/file/dw-kap5.pdf
5 Spatial Access Methods

• Example of point-shaped objects: weather stations
  – Attributes
    • Position
    • Owner
    • Year of construction
    • Analog/digital

http://de.wikipedia.org/
• Sorting points by Z-order
  – Recursive decomposition of space in quadrants until every quadrant contains at most one point
• Sorting points by Z-order
  – Recursive decomposition of space in quadrants until every quadrant contains at most one point
• Sorting points by Z-order
  – Recursive decomposition of space in quadrants until every quadrant contains at most one point
• Sorting points by Z-order

  - Recursive decomposition of space in quadrants until every quadrant contains at most one point

⇒ B, C, A, F, D, E, G, J, H, K, I
• A more realistic example
• An even more realistic example (z-order of an ALKIS inventory data extract)
• Efficient calculation of the Z-value by bit-interleaving

– The Z-value of a quadrant consists of alternating bits from the binary representations of its x and y coordinate values, i.e.:

<table>
<thead>
<tr>
<th>x-coordinate</th>
<th>y-coordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td>001111</td>
<td>01010</td>
</tr>
</tbody>
</table>

Z-value

\[
\begin{array}{cccc}
11 & 0101 & 0111 & 1101 & 1111 \\
10 & 0100 & 0110 & 1100 & 1110 \\
01 & 0001 & 0011 & 1001 & 1011 \\
00 & 0000 & 0010 & 1000 & 1010 \\
\end{array}
\]
Example: range query

- Calculate Z-values of the query window: Min=16, Max=30
- Search for Min
- Check all points whose Z-values ≤ Max
5 Spatial Access Methods

• Problems
  – Only suitable for points
  – False positives, the solutions supplied by the index have to be verified
  – Adequacy depends on the position of the query window
Indexes can be classified according to

- Data types, which can be indexed
  - Points (Point Access Method) or extended objects (Spatial Access Method)

- Principle of space decomposition:

<table>
<thead>
<tr>
<th></th>
<th>data-driven (tree)</th>
<th>space-driven (trie)</th>
</tr>
</thead>
<tbody>
<tr>
<td>structure dependant on the insertion order</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>indexing dead space</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>guaranteed storage utilization</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
5.1 Quadtree

• Main principle:
  – Successive decomposing of space into quadrants
  – Search tree whose nodes have four children each

• Decomposition may be space- or data-driven

• Can be used for points as well as for extended objects

• Not balanced, can degenerate
5.1 Quadtree

- **Point quadtree (data-driven)**
  - Indexing of points
  - Data pointers in leaf and internal nodes
  - Insertion
    - Search ends at Nil node, replace this node with the new node containing the element
  - Deletion of internal nodes difficult
    - Remove the subtree whose root is the node to be deleted
    - Re-insert nodes of the subtree
5.1 Quadtree

– Example: insertion
5.1 Quadtree

– Example: insertion
5.1 Quadtree

– Example: insertion
5.1 Quadtree

– Example: insertion
5.1 Quadtree

– Example: insertion
5.1 Quadtree

- Range query: Min(5, 16), Max(20, 19)
  - 4 comparisons per node
    - Min.x > x → skip w
    - Min.y > y → skip s
    - Max.x < x → skip e
    - Max.y < y → skip n
5.1 Quadtree

– Unfavorable insertion order
5.1 Quadtree

• Disadvantages
  – High storage usage
    • 4 Nil nodes per leaf at least, additional ones may be in internal nodes
  – Deletion expensive
  – $k$ comparisons for point queries, $2^k$ for range queries per node
  – Variants: Pseudo-Quadtree
    • Using arbitrary points for the partitioning process
    • Data pointers only in leaf nodes
5.1 Quadtree

- Point quadtree (space-driven)
  - Recursive subdivision of the space into quadrants until every quadrant contains at most the given number of points
  - Extent of space has to be known
5.1 Quadtree

- Internal nodes contain only node pointer
- Data pointer only in leaf nodes
- Structure depends on the location of points
- Insertion
  - Search ends at Nil node: insert new leaf node
  - Search ends at leaf node: replace leaf with a new internal node, insert both points as new leaves
- Deletion
  - Only leaves are deleted
  - If an internal node has only one child after deletion it is replaced by its child
5.1 Quadtree

- Region quadtree (space-driven)
  - Indexing of surfaces
  - Decomposition of space until the quadrants are empty or completely covered by an object
5.2 R-tree

- The R-tree is the prototype of a multidimensional extension of the B-tree
  - The root has at least two child nodes
  - Every internal node has got between $m$ and $M$ child nodes (multiway tree)
  - $M$ and $m \leq M$ are predisposed
  - For each entry of an internal node the MBR containing all the rectangles of its child nodes is stored
5.2 R-tree

- Overlapping clusters (the more overlap the less efficient)
- Objects are stored in leaves, internal nodes for navigation only
- Height balanced tree (i.e. its subtrees differ in height by no more than one and the subtrees are height-balanced, too)
- Dynamic index structure (supports insert, update and delete)

http://www.geoinformatik.uni-rostock.de/
5.2 R-tree

– Construction
5.2 R-tree

• Search
  – Recursively from the root to the leaves
    • One path at a time
    • If the object has not been found in this subtree, the next search path will be traversed
  – The traversal order of paths is arbitrary
  – Good performance is not guaranteed
  – In the worst case all paths have to be searched (because of overlap)
  – Search algorithms try to skip as many irrelevant subtrees as possible (pruning)
5.2 R-tree

– Search in internal node:
  • Check for each entry if its MBR intersects S
  • For all intersecting entries continue searching in all its child nodes

– Search in leaf nodes:
  • Check for each entry if its MBR intersects S
  • For all intersecting entries check if the original objects satisfy the search condition, if so they belong to the solution

– Search algorithms for R-trees are the most efficient if overlap and coverage are minimized
Example:

- Example:
5.2 R-tree

• Insertion
  – Search the best leaf node, according to spatial criteria (ChooseLeaf)
  – If the node contains less than M entries, insert the object there
  – Otherwise overflow handling is necessary and the node is split (SplitNode), so that overlap and coverage are minimized
  – MBR of the parent node has to be enlarged if the new object is not entirely inside (AdjustTree)
  – If the root is splitted create a new root whose children are the splitted nodes of the old root
5.2 R-tree

- Insert $k$ either into $C$ or into $D$
- If $D$ is chosen the added area is bigger, but there is no overlap
5.2 R-tree

– Heuristic
  • Always insert a new object into the node resulting in the smallest increase in volume
  • If it lies inside a MBR no enlargement is necessary
  • If there are more than one possibility, choose the node with the smallest volume

– Overflow
  • If a new entry should be inserted into a full node than $M+1$ entries have to be distributed to two nodes
  • During subsequent searches it should not be necessary to search both subtrees
  • The smaller the MBRs the less overlap
5.2 R-tree

- Calculate the sum of the area of both rectangles and minimize the dead space
- Decision how to split is not trivial
- There are many possibilities to distribute the objects to two MBRs
- The volume of both MBRs should be as small as possible
- Compare all possible distributions takes exponential time (practically impossible)
5.2 R-tree

– Overflow handling with a quadratic complexity

• For each pair of objects calculate the area of their MBR and choose the pair with the largest MBR
• As these objects should not belong to the same node they are used as seed points of two new nodes
• Calculate for all other objects the difference in increase in volume in respect of both MBRs
• Insert the object with the largest difference into the adequate MBR and update it
• Repeat this process for the remaining objects
5.2 R-tree

- Determining the seeds with a linear complexity
  - For each dimension:
    - Find the rectangle with the highest low edge and the rectangle with the lowest high edge
    - Calculate the distance between them and normalise it by dividing it by the width of the MBR of the overflowing node
  - The final seeds are the two objects having the greatest normalized separation

- This linear cost algorithm is usually sufficient as the quality of the splits i.e. the minimality of overlaps is similar to that of the quadratic cost algorithm
5.2 Example Overflow Handling
5.2 Example Overflow Handling

• X-direction:
  – Choose e and j
  – \( d_x = \frac{2}{11} \approx 0.18 \)

• Y-direction:
  – Choose m and j
  – \( d_y = \frac{3.5}{9} \approx 0.43 \)

• As \( d_y > d_x \) choose m and j
5.2 Example Overflow Handling

- **Insert g**
  - to m: 24
  - to j: 41.5
  - Difference: 17.5

- **Insert e**
  - to m: 14
  - to j: 52
  - Difference: 38
5.2 Example Overflow Handling

- Result
5.2 R-tree

- **Deletion**
  - Search for the leaf containing the entry (*FindLeaf*)
  - Delete the entry from the leaf (*DeleteRecord*)
  - Condense the tree if the node contains less than m entries, i.e. the node is deleted completely and the other entries are reinserted (*CondenseTree*)
  - If only one child of the root is left this child becomes the new root

- **Updates**
  - If an object is updated its MBR might change
  - In that case the corresponding entry has to be deleted, updated and reinserted
5.2 R-tree

- $R^+$-tree
  - Overlap of adjacent MBRs is forbidden
  - Several leaves may contain an entry for one object
  - Search is more efficient, but its scalability is similar to R-trees
5.2 R-tree

- Differences between $R^+$-trees and R-trees
  - Insertion: object may be inserted in more than one leaf
  - Insertion: propagation of splits down the tree as overlaps are not allowed
  - Deletion: there is no minimum number of child nodes anymore

- The biggest advantage of $R^+$-trees is the improved search efficiency as only a single path has to be searched for a point query
- A disadvantage is that nodes are often under-filled due to many splits
- $R^+$-trees often degenerate in case of many updates
5.3 K-d tree

- K-dimensional, binary search tree
- On every level the space is decomposed perpendicular to one dimension → one comparison
- Indexing of points
- Variant for extended objects: Bintree
- Not balanced
- A K-d tree for n points
  - Needs $O(n)$ space
  - May be constructed in $O(n \log n)$ time
5.3 K-d tree

- Example: data driven, alternating splits in x- and y-dimension

Spatial Databases and GIS – Karl Neumann, Sarah Tauscher – Ifis – TU Braunschweig
5.3 K-d tree

- Split at a data points or at an arbitrary points
- Choice of split dimension
  - Alternating or dimension with the largest extent
- Choice of split position
  - Median or average
5.3 K-d tree

• Determination of the split plane
  – Calculation of the median of n elements necessary
  – Complex algorithm to determine the median in $O(n)$ or:
    • Sort all points in two lists one according to the x- and one to the y-coordinate
    • After a split these lists are splitted respectively $\rightarrow$ traversal in linear time possible
    • Determining the median in $O(1)$ and creation of sublists in $O(n)$ time
• Bintree
  – Like the region quadtree, but the space is not divided into quadrants but into half-spaces
5.3 K-d tree

• Deletion
  – Similar to binary search tree: the node to delete is replaced by its immediate predecessor or successor
  – Problem: Several different orderings of the elements (one for each dimension)
  – Solution: Search the node which is the immediate predecessor or successor with respect to the discriminant of the node to delete
5.3 K-d tree

• Deletion
  – Similar to binary search tree: the node to delete is replaced by its immediate predecessor or successor
  – Problem: Several different orderings of the elements (one for each dimension)
  – Solution: Search the node which is the immediate predecessor or successor with respect to the discriminant of the node to delete
5.4 BSP tree

• Binary Space Partitioning (BSP)
  – Recursive bipartitioning of the n-dimensional space by (arbitrary) (n-1)-dimensional hyperplanes (in 2d: lines) → generalized k-d tree

• Usually used to index surfaces or lines

• Class variables of a BSP tree node

```java
class Node{
    Hyperplane ph;  // split plane (instead of the discriminant)
    Node leftChild; // left child, is „in front of“ the split plane
    Node rightChild; // right child, is „behind“ the split plane
    Set oSet;       // set of objects in the plane
}
```
5.4 BSP tree

• **Pseudocode for the construction of a BSP tree**

```java
Node buildBSPtree(o) {                      // o is set of objects
    if o.isEmpty() return null;
    Node N = new Node();
    N.ph = choosePartitioningHyperplane(o);
    Set fSet = new Set();                   // to save the objects in front of ph
    Set bSet = new Set();                   // to save the objects behind ph
    for each Object z in o do{
        if z is coincident to ph, add z to N.oSet;
        if z is in front of ph, add z to fSet;
        if z is behind ph, add z to bSet;
        if z spans ph, split z and add pieces to fSet and bSet;
    }
    N.leftChild = buildBSPTree(fSet);
    N.rightChild = buildBSPTree(bSet);
    return N;
}
```
• Determining the position of an object z relative to a split plane g

  - 2-dimensional split line: \(ax + by + c = 0\)
  - Calculate \(ax + by + c\) for all points of z

• If all values < 0
  z „is in front of“ g

• If all values > 0
  z „is behind“ g

• If all values = 0
  z lies on g

• Otherwise z intersects the split line and has to be splitted
### 5.4 BSP tree

- **Goals for the choice of split planes**
  - Balanced tree
  - Avoid splitting of objects
  - Split planes orthogonal to the axes

- **Heuristic to estimate the costs of a split plane**

  
  \[
  \text{Difference between the number of objects in front of and behind the split plane} + 15 \times \text{number of splitted objects} + 5 \text{ if split plane is not orthogonal to one of the axes}
  \]
A split plane which do not divide any objects is called Free Partition

- If a part of a boundary of an object which already intersects a BSP region is chosen as a split plane, it is a free partition in respect of that region.
- Under the assumption that there are no intersections between objects.
5.4 BSP tree

- Application in computer graphics
  - E.g. visibility computation, ray tracing
  - Surfaces on the same side as the observer
    - May cover surfaces on the other side
    - Cannot be covered by surfaces on the other side
  - Determination of the drawing order
    - Back-to-front: background objects are drawn first and are painted over by objects in the foreground
      → wanted: ordering of the objects with respect to their „relative distance“ from the observer
5.4 BSP tree

- For each node, determine on which side of it the observer lies

- Inorder traversal defines the order of the objects
  1. Subtree on the opposite side from the observer
  2. Node
  3. Subtree on the same side as the observer

- Inorder traversal on a binary search tree
  1. Left subtree
  2. Node
  3. Right subtree
5.4 BSP tree

• Example
  – Construction of a BSP tree by a sequence of insertion operations (without application of the heuristic)
    • 3-dimensional objects (upright walls)
5.4 BSP tree

• Example

– Construction of a BSP tree by a sequence of insertion operations (without application of the heuristic)
  • 3-dimensional objects (upright walls)
• Example
  – Construction of a BSP tree by a sequence of insertion operations (without application of the heuristic)
    • 3-dimensional objects (upright walls)
5.4 BSP tree

• Example
  – Construction of a BSP tree by a sequence of insertion operations (without application of the heuristic)
    • 3-dimensional objects (upright walls)
• Example
  – Construction of a BSP tree by a sequence of insertion operations (without application of the heuristic)
  • 3-dimensional objects (upright walls)
• **Example**
  
  - Construction of a BSP tree by a sequence of insertion operations (without application of the heuristic)
  
  • 3-dimensional objects (upright walls)
5.4 BSP tree

• Example
  – Construction of a BSP tree by a sequence of insertion operations (without application of the heuristic)
  • 3-dimensional objects (upright walls)
5.4 BSP tree

Drawing order: 3, 4b, 5b, 0

5.4 BSP tree

Drawing order: 3, 4b, 5b, 0, 2, 6ff

5.4 BSP tree

Drawing order: 3, 4b, 5b, 0, 2, 6ff, 4f, 6fb, 1

5.4 BSP tree

Drawing order: 3, 4b, 5b, 0, 2, 6ff, 4f, 6fb, 1, 5f, 6b

5.5 Grid file

- Main idea:
  - $k$-dimensional, dynamic array as index (on disk)
  - Index contains (except for management information where necessary) only pointers to data pages (buckets)
  - Data is stored in (big) buckets

- Two disk accesses for exact match queries:
  - Independent of the distribution of values and the number of stored objects

- For points only

- Data-driven partitioning of space
5.5 Grid file

- Components:
  - K scales $S_i$ (with $i$ from 1 to $k$) define the grid on the $k$-dimensional data space $D$
  - Cell or grid directory (GD): dynamic $k$-dimensional matrix to map $D$ on the buckets
  - Bucket: storage of the points of one or more cells (bucket region BR)
Properties

- 1:1-relation between cells and elements of GD
- Element of GD = bucket pointer
- n:1-relation between cells and buckets
- Cells have to form a rectangle in order to be assigned to the same bucket
5.5 Grid file

- Exact match query
  - Search the scales (binary search) for the position of the grid cell that covers the point (→ no disk access, if the scales fit into main memory)
  - Read the grid directory to identify the bucket (→ first disk access)
  - Read the bucket (→ second disk access)
5.5 Grid file

• Insertion
  – Search for the point to be inserted and if possible put it in the found bucket
  – In case of bucket overflow
    • Add new bucket, update GD
    • If a grid refinement is necessary, update scales
    • Choice of dimension for the grid refinement e.g. alternating or “longest” interval
    • Choice of split position e.g. middle or local median
Example: insertion with bucket overflow without grid refinement
5.5 Grid file

– Example: insertion with bucket overflow without grid refinement
Example: insertion with bucket overflow and grid refinement
Example: insertion with bucket overflow and grid refinement
Example: insertion with bucket overflow and grid refinement
5.5 Grid file

• Deletion
  – **Merging of buckets**
    • Ideally the buckets to be merged were split at some earlier point and none of them was split again → buddy system
    • With adjacent cells, complex
    • Not always possible
  – **In general grid refinements will not be revoked**
5.6 Summary

- **One-dimensional orderings**
  - Impose a linear ordering on multidimensional points
  - Z-order, Hilbert's curve, Gray code

- **Quadtree**
  - Point quadtree: construction, range query
  - Region quadtree

- **R-tree**
  - Construction
  - Search
  - Insertion, overflow handling
  - $R^+$-tree
5.6 Summary

- **K-d tree**
  - Choice of split plane
  - Deletion
  - Bintree

- **BSP tree**
  - Data structure
  - Choice of split plane
  - Rendering

- **Grid file**
  - Insertion, overflow handling
  - Deletion
5.6 Summary

GIS

collect
manage
analyze
visualize

indexes

objects
geometry
vector

query evaluation
approximations