Indexing: ordered indexes, continued

Lecture 3
2ID35, Spring 2015

George Fletcher

Faculteit Wiskunde & Informatica
Technische Universiteit Eindhoven

29 April 2015

Admin

Project part 2 will be posted later today. Once posted:

- find your team mates
- study your paper
- determine your research problem
- first report due on or before 10 May

Where we've been

Last time

- File structures

Where we've been

Last time

- File structures
- I/O model of computation

Where we've been

Last time

- File structures
- I/O model of computation
- external sorting

Where we've been

Last time

- File structures
- I/O model of computation
- external sorting
- B+ trees

Today's agenda

Ordered indexes, continued

- R trees
- GiST: generalized search trees

Ordered indexing: spatial data

Why spatial/multi-dimensional data?

- GIS, CAD design, multimedia, ...
- composite search keys are not enough

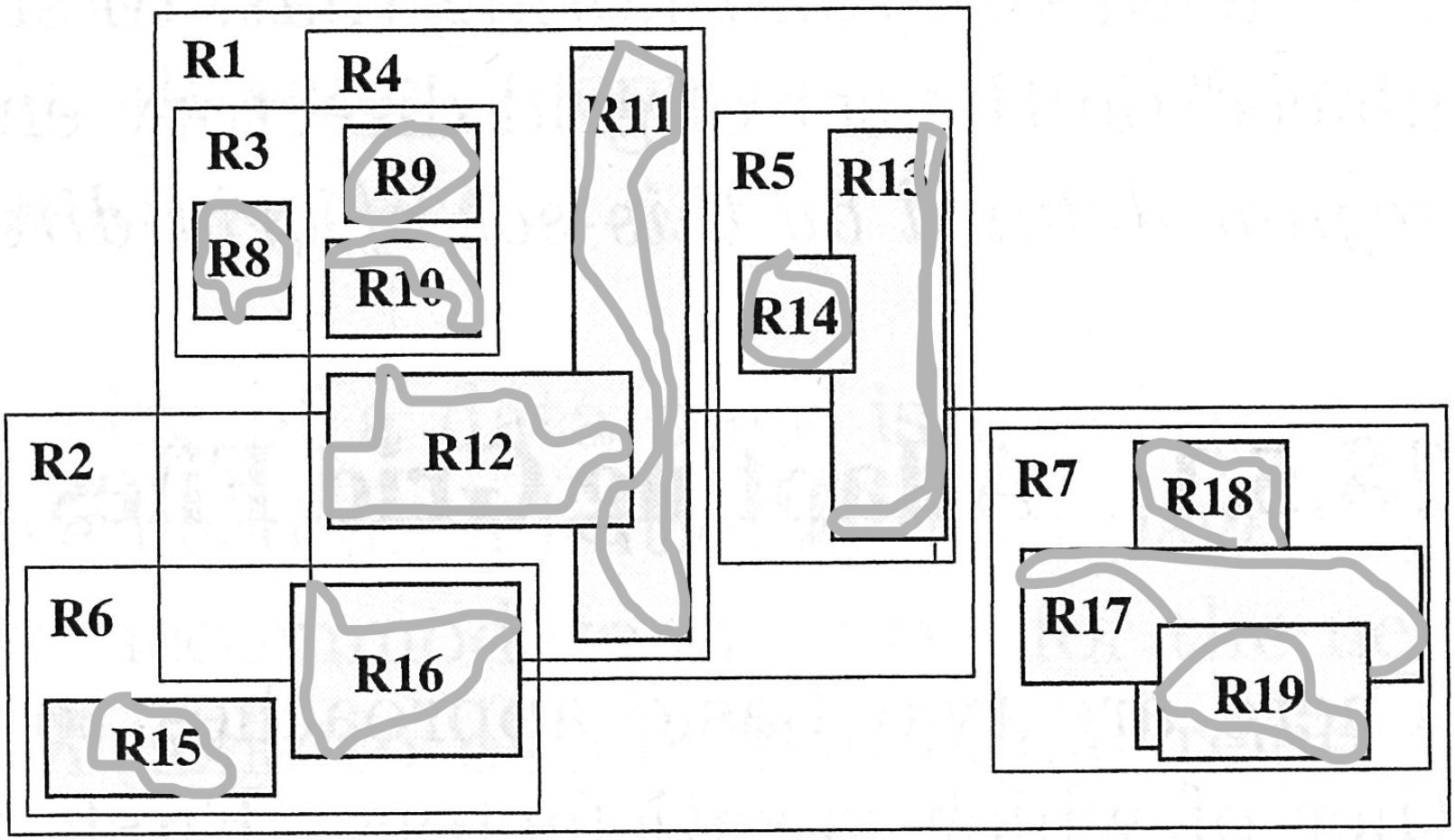
Ordered indexing: R-trees

- height-balanced, dynamic search tree
- designed to minimize I/O

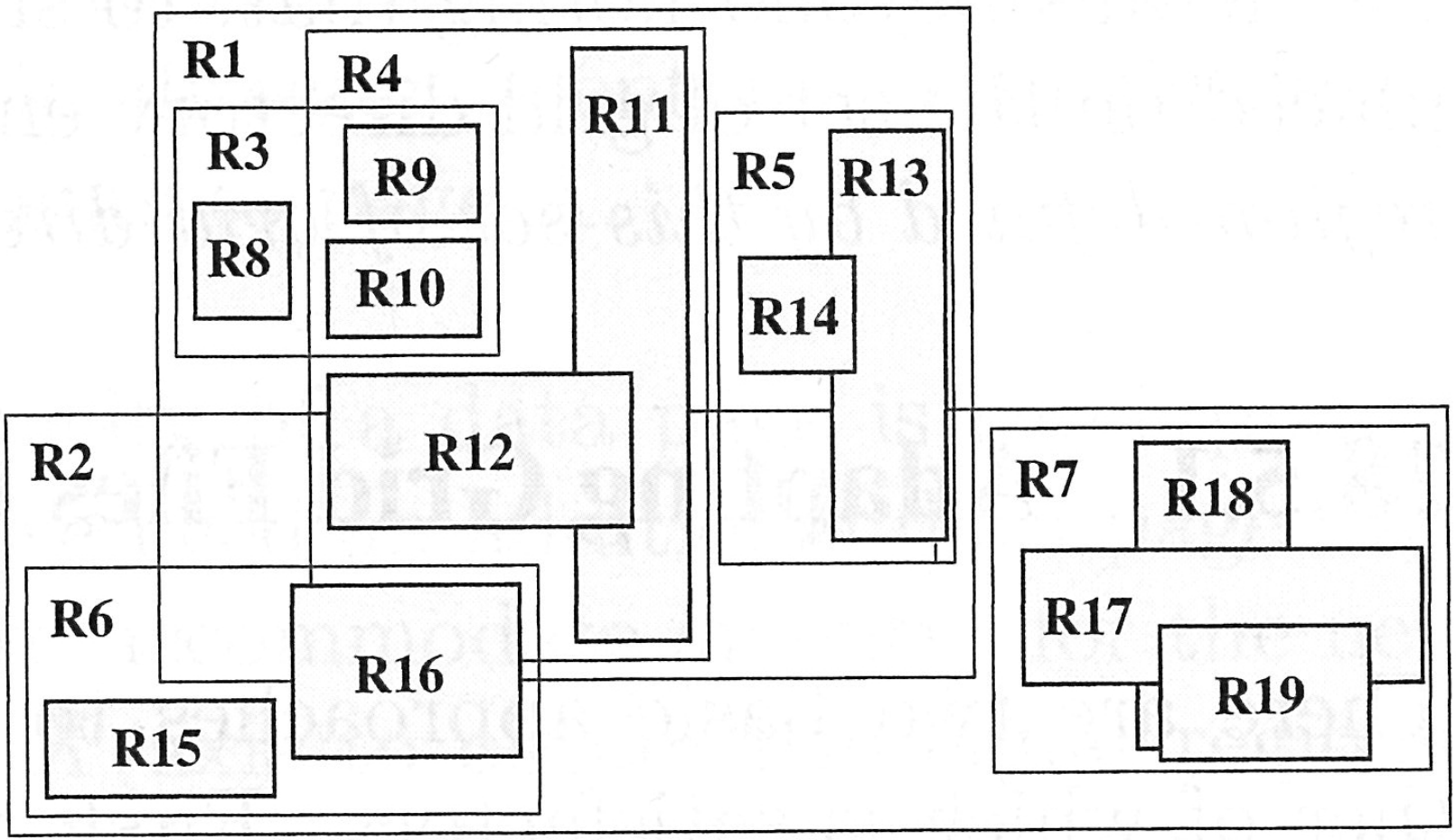
Ordered indexing: R-trees

- height-balanced, dynamic search tree
- designed to minimize I/O
- instead of range of values, a rectangular bounding box is associated with each node
- efficient spatial search

Ordered indexing: R-trees



Ordered indexing: R-trees



R-trees: structure

Properties

- leaf entry:

  [*n-dimensional bbox, pointer to record*]

  bounding box is tightest bounding box for data object

R-trees: structure

Properties

- leaf entry:

  [*n-dimensional bbox, pointer to record*]

  bounding box is tightest bounding box for data object
- internal entry:

  [*n-dimensional bbox, pointer to child node*]

  the box covers all boxes in subtree rooted at child

R-trees: structure

Properties

- leaf entry:

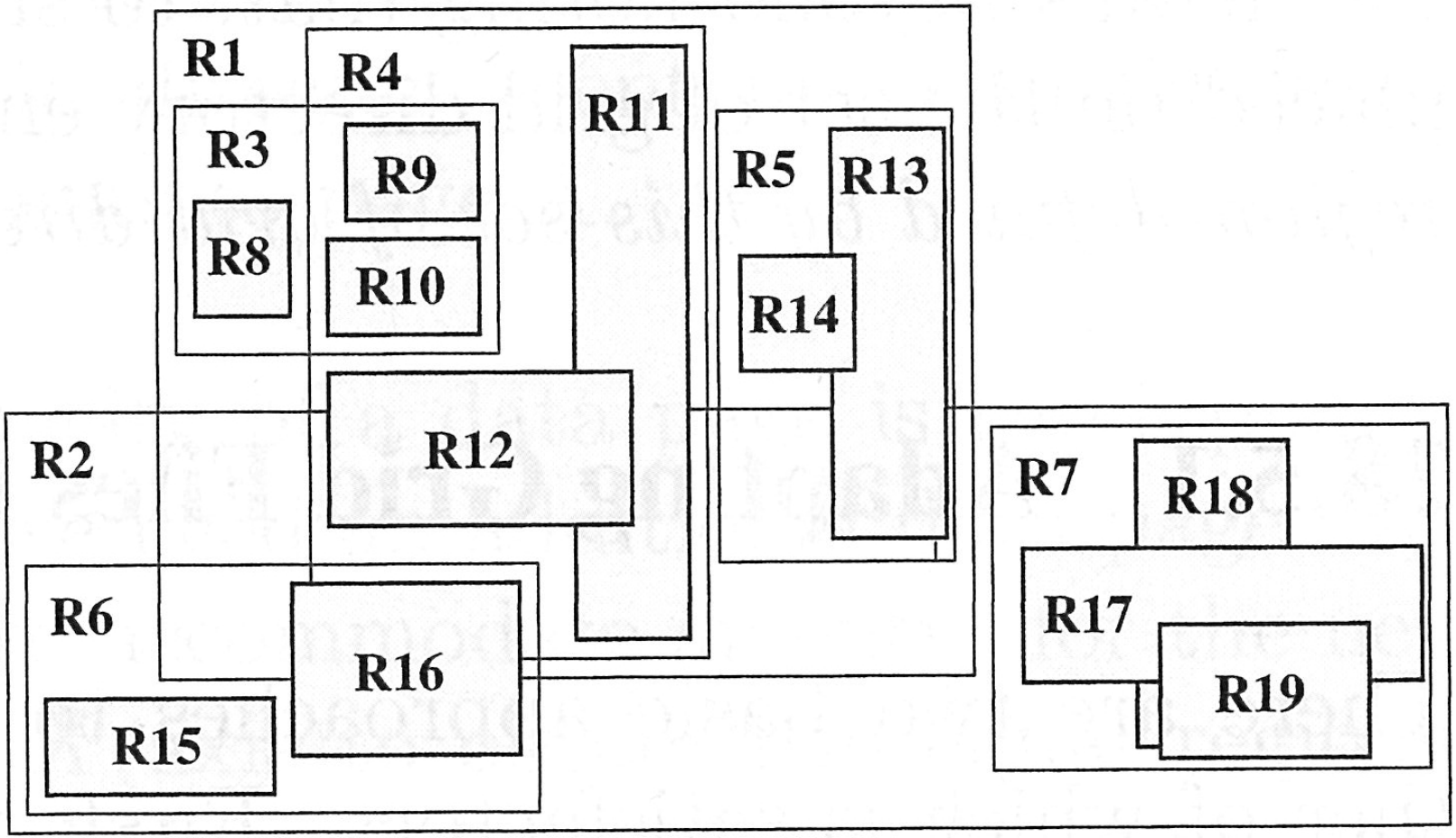
  [*n-dimensional bbox, pointer to record*]

  bounding box is tightest bounding box for data object
- internal entry:

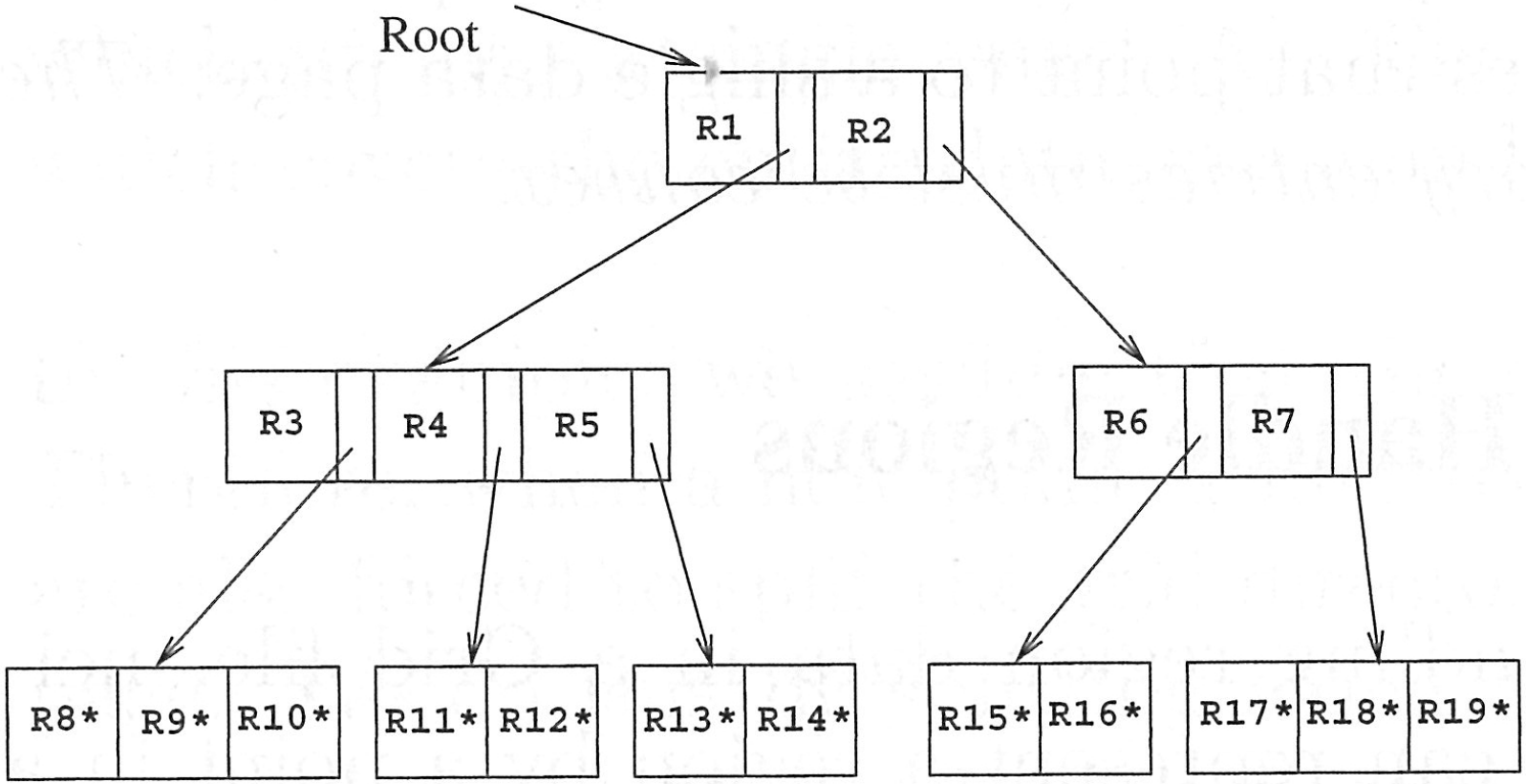
  [*n-dimensional bbox, pointer to child node*]

  the box covers all boxes in subtree rooted at child
- same minimal occupancy rules as for B+ trees

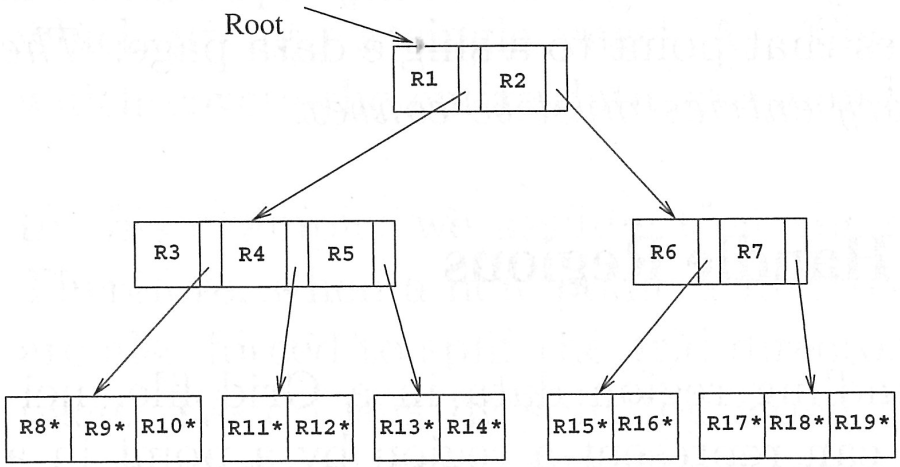
R-trees: structure

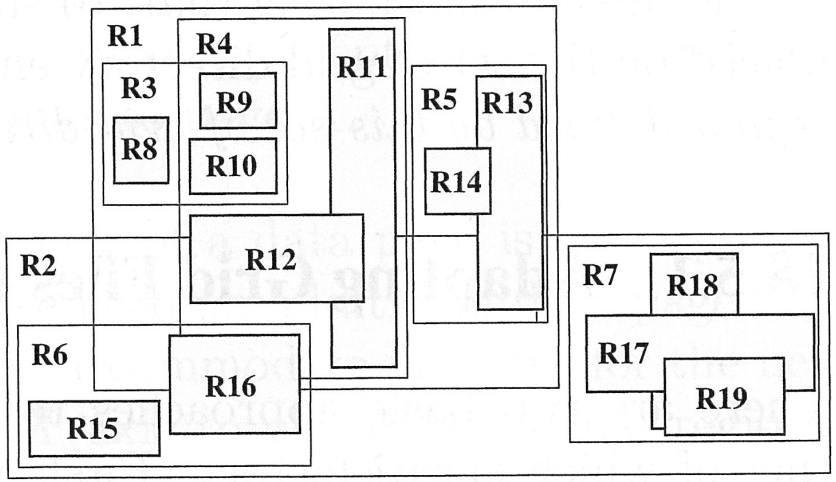


R-trees: structure

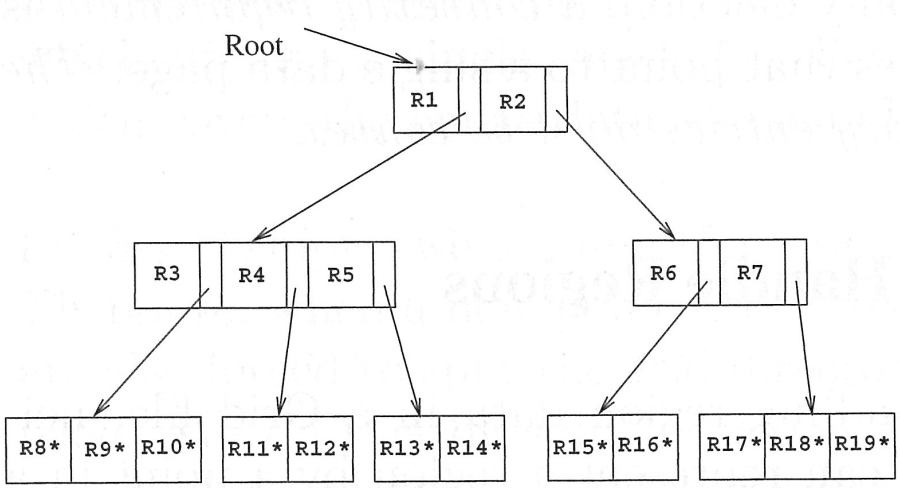


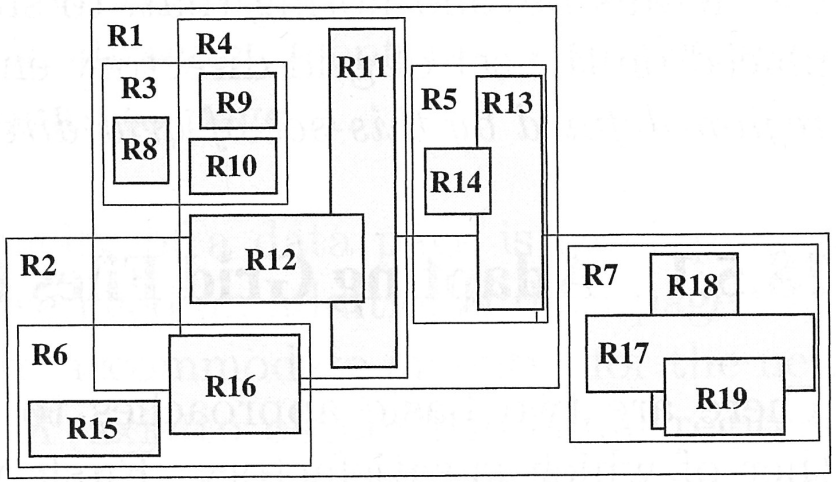
R-trees: structure





R-trees: structure





Note that, unlike B+ trees, we have no ordering on objects (and hence no ordering on tree nodes, siblings, etc.)

R-trees: search

Search for objects overlapping box B

- if current node is internal
  - for each entry [E, *pointer*], if E overlaps B, search *pointer*

R-trees: search

Search for objects overlapping box B

- if current node is internal
  - for each entry $[E, \textit{pointer}]$, if E overlaps B, search *pointer*
- if current node is leaf
  - for each entry $[E, \textit{rid}]$, if E overlaps B, *rid* identifies an object which might overlap B

R-trees: search

Search for objects overlapping box B

- if current node is internal
  - for each entry [E, *pointer*], if E overlaps B, search *pointer*
- if current node is leaf
  - for each entry [E, *rid*], if E overlaps B, *rid* identifies an object which might overlap B

Note that, unlike B+ tree, we may have to search multiple subtrees

R-trees: insertion

Insert $[B, rid]$ in node N

- if N is internal
  - Insert $[B, rid]$ in child node C, where the bounding box associated with C needs the least enlargement to cover B

R-trees: insertion

Insert $[B, rid]$ in node N

- if N is internal
  - Insert $[B, rid]$ in child node C, where the bounding box associated with C needs the least enlargement to cover B
- if N is leaf
  - if N has space, insert
  - else, split N into N_1 and N_2, adjust N's parent to only cover new N_1, and add entry for N_2

R-trees: deletion

Delete is handled as expected, except that under-full nodes are just deleted, and the remaining entries re-inserted.

R-trees

R-trees

- Very popular spatial indexing mechanism (MySQL, Oracle, Postgres, ...)
- Intensively studied, leading to many variants such as R* and R+ trees
  - example: Hilbert R-trees

R-trees: Hilbert variant

Hilbert R-trees

- lack of ordering has negative impact on clustering (i.e., spatial locality) of objects
  - leads to poor space utilization (~70%)

R-trees: Hilbert variant

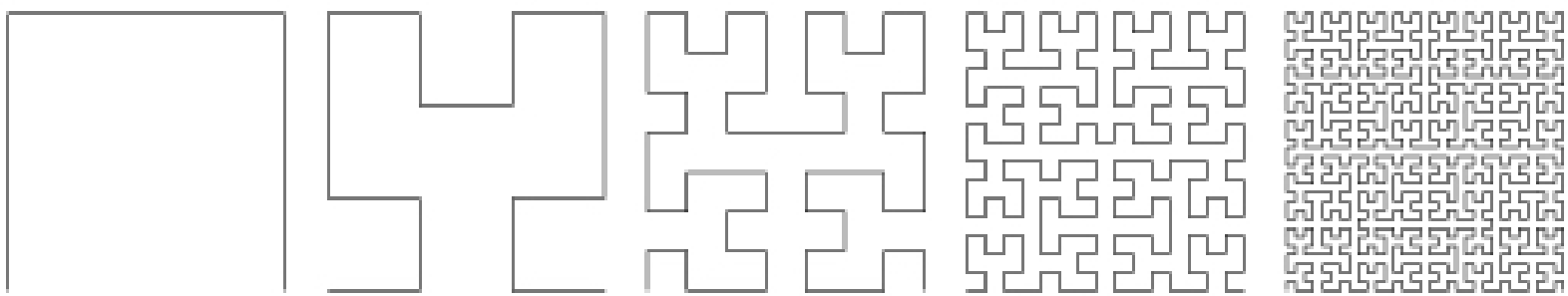
Hilbert R-trees

- lack of ordering has negative impact on clustering (i.e., spatial locality) of objects
  - leads to poor space utilization (~70%)
- *key idea of Hilbert R-trees:* add an ordering to objects (and hence on nodes in the tree)
  - leads to excellent space utilization (~100%) and search/update costs (up to ~30% fewer page accesses)

R-trees: Hilbert variant

Hilbert curves

- example of a "space filling" curve, proposed by Hilbert in 1891
- imposes a linear ordering on the points in the unit n-cube
- has very good spatial locality



R-trees: Hilbert variant

Ordering bounding boxes with Hilbert curves

- identify each bounding box with the Hilbert curve value of the center point of the box
  - i.e., its position on the curve

R-trees: Hilbert variant

Ordering bounding boxes with Hilbert curves

- identify each bounding box with the Hilbert curve value of the center point of the box
  - i.e., its position on the curve
- extend each entry in non-leaf nodes to also include the largest Hilbert value (LHV) among the boxes contained in the subtree rooted at the entry

  [*LHV, n-dimensional bbox, pointer to child node*]

R-trees: Hilbert variant

Ordering bounding boxes with Hilbert curves

- identify each bounding box with the Hilbert curve value of the center point of the box
  - i.e., its position on the curve
- extend each entry in non-leaf nodes to also include the largest Hilbert value (LHV) among the boxes contained in the subtree rooted at the entry

  [*LHV, n-dimensional bbox, pointer to child node*]
- Search as usual; insertion/deletion are guided by Hilbert value of the inserted/deleted box
  - also, Hilbert ordering can be effectively used for bulk-loading/building the index

Search trees, in general

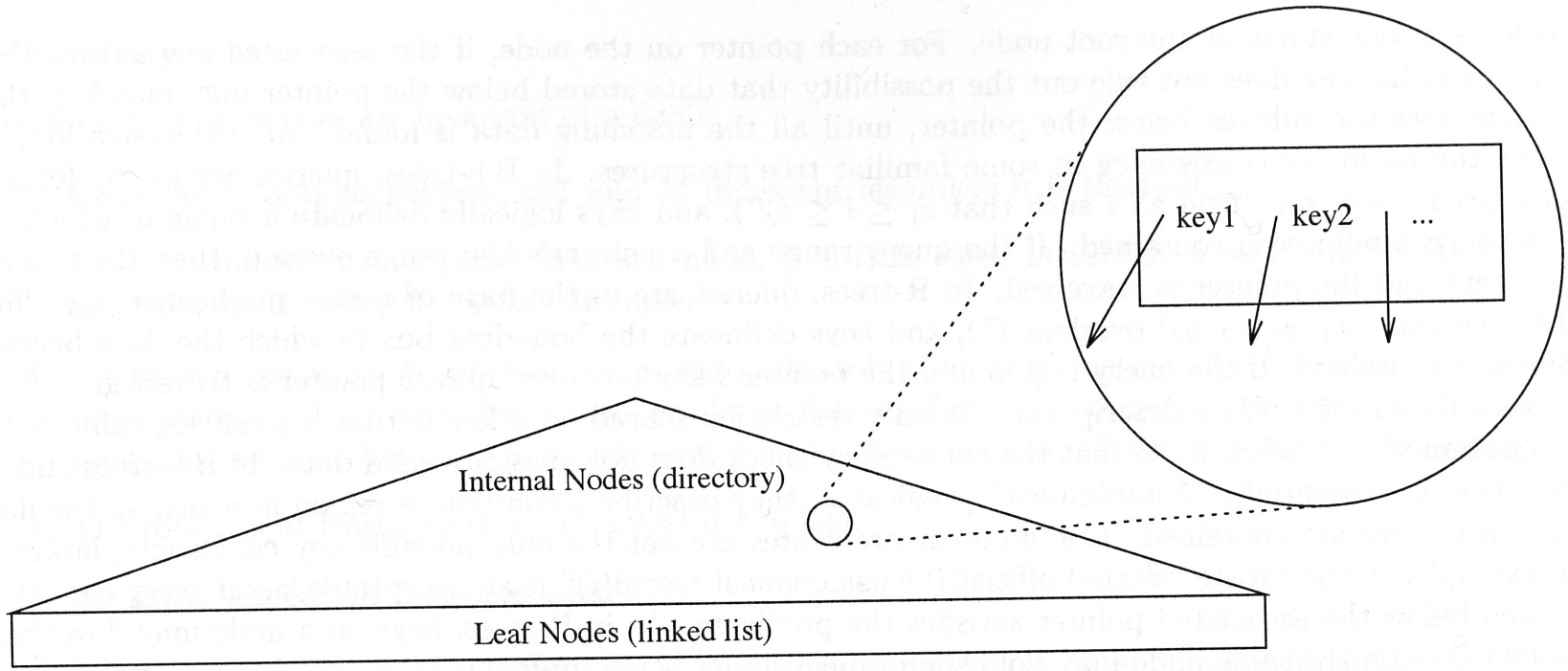
There has been an explosion in types of data: geographic data, multimedia, CAD, document libraries, sequence data, fingerprint data, biochemical and bioinformatics data, ...

Search trees, in general

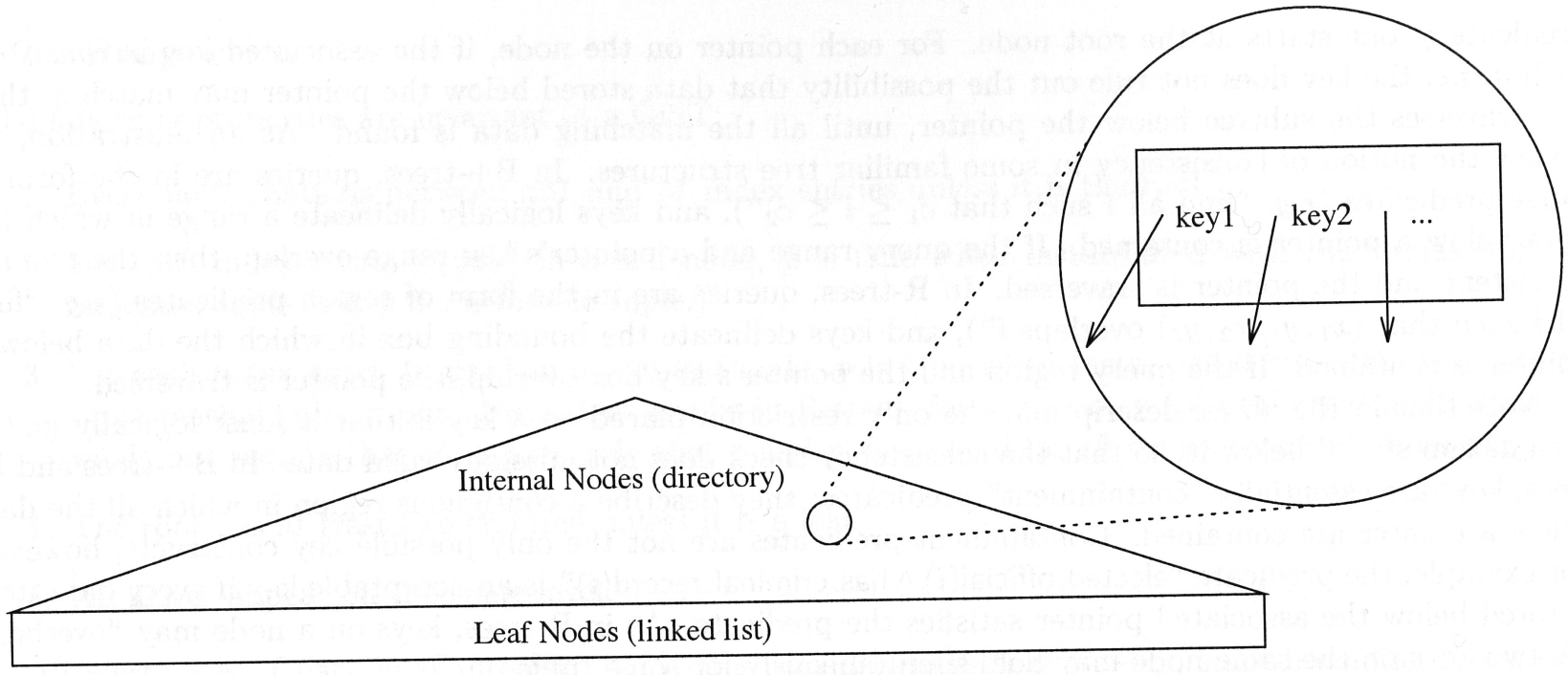
Consequently, there has been an explosion of tree-based indexing techniques ...

often based on specialized search trees, or search trees over extensible data types.

Search trees, in general



Search trees, in general



What can we say in general about tree-based ordered indexes? How far can we generalize this structure?

GiST: generalized search trees

Proposal for generic tree-index infrastructure

- a *search key* may be any arbitrary predicate q that holds for each datum below the key

GiST: generalized search trees

Proposal for generic tree-index infrastructure

- a *search key* may be any arbitrary predicate q that holds for each datum below the key
- a *search tree* is a hierarchy of partitions of a dataset, in which each partition has a categorization that holds for all data in the partition.

GiST: generalized search trees

Proposal for generic tree-index infrastructure

- a *search key* may be any arbitrary predicate q that holds for each datum below the key
- a *search tree* is a hierarchy of partitions of a dataset, in which each partition has a categorization that holds for all data in the partition.

Keys are user-defined (e.g., integers, bounding boxes, etc.).

GiST: generalized search trees

Proposal for generic tree-index infrastructure

- a *search key* may be any arbitrary predicate q that holds for each datum below the key
- a *search tree* is a hierarchy of partitions of a dataset, in which each partition has a categorization that holds for all data in the partition.

Keys are user-defined (e.g., integers, bounding boxes, etc.).

Available as part of PostgreSQL

GiST: generalized search trees

Properties

- every node has a minimal occupancy

GiST: generalized search trees

Properties

- every node has a minimal occupancy
- for each index entry (p, ptr) in a leaf node, p is true when instantiated with the values from the tuple indicated by *ptr*

GiST: generalized search trees

Properties

- every node has a minimal occupancy
- for each index entry (p, ptr) in a leaf node, p is true when instantiated with the values from the tuple indicated by ptr
- for each index entry (p, ptr) in an internal node, p is true when instantiated with the values from any tuple reachable from ptr

GiST: generalized search trees

Properties

- every node has a minimal occupancy
- for each index entry (p, ptr) in a leaf node, p is true when instantiated with the values from the tuple indicated by ptr
- for each index entry (p, ptr) in an internal node, p is true when instantiated with the values from any tuple reachable from ptr
- the root has at least two children unless it is a leaf
- all leaves appear on the same level

GiST: generalized search trees

Key designer must define the following methods

- Consistent(E, q), for entry $E = (p, ptr)$ and search predicate q

GiST: generalized search trees

Key designer must define the following methods

- Consistent(E, q), for entry $E = (p, ptr)$ and search predicate q
- Union(P), for list of entries P, return a new predicate satisfied by all members of P

GiST: generalized search trees

Key designer must define the following methods

- Consistent(E, q), for entry $E = (p, ptr)$ and search predicate q
- Union(P), for list of entries P, return a new predicate satisfied by all members of P
- Compress(E) and Decompress(E), (de)compressed representation for p, for insertion/reading of E into/from a node

GiST: generalized search trees

Key designer must define the following methods

- Consistent(E, q), for entry $E = (p, ptr)$ and search predicate q
- Union(P), for list of entries P, return a new predicate satisfied by all members of P
- Compress(E) and Decompress(E), (de)compressed representation for p, for insertion/reading of E into/from a node
- Penalty(E_1, E_2), for inserting E_2 into subtree at E_1

GiST: generalized search trees

Key designer must define the following methods

- Consistent(E, q), for entry $E = (p, ptr)$ and search predicate q
- Union(P), for list of entries P, return a new predicate satisfied by all members of P
- Compress(E) and Decompress(E), (de)compressed representation for p, for insertion/reading of E into/from a node
- Penalty(E_1, E_2), for inserting E_2 into subtree at E_1
- PickSplit(P), into two new lists P_1 and P_2, maintaining occupancy requirements

GiST: generalized search trees

Basic infrastructure:

- Search
- Search on linearly ordered domains
  - using FindMin and Next

GiST: generalized search trees

Basic infrastructure:

- Search
- Search on linearly ordered domains
  - using FindMin and Next
- Insert
  - using ChooseSubTree, Split, and AdjustKeys

GiST: generalized search trees

Basic infrastructure:

- Search
- Search on linearly ordered domains
  - using FindMin and Next
- Insert
  - using ChooseSubTree, Split, and AdjustKeys
- Delete

GiST: search

Algorithm Search(R, q)

Input: GiST rooted at R, predicate q

Output: all tuples that satisfy q

Sketch: Recursively descend all paths in tree whose keys are consistent with q.

S1: [Search subtrees] If R is not a leaf, check each entry E on R to determine whether Consistent(E, q). For all entries that are Consistent, invoke Search on the subtree whose root node is referenced by E.`ptr`.

S2: [Search leaf node] If R is a leaf, check each entry E on R to determine whether Consistent(E, q). If E is Consistent, it is a qualifying entry. At this point E.`ptr` could be fetched to check q accurately, or this check could be left to the calling process.

GiST: search on linearly ordered domains

Algorithm FindMin(R, q)

Input: GiST rooted at R, predicate q

Output: minimum tuple in linear order that satisfies q

Sketch: descend leftmost branch of tree whose keys are Consistent with q. When a leaf node is reached, return the first key that is Consistent with q.

FM1: [Search subtrees] If R is not a leaf, find the first entry E in order such that Consistent(E, q)[1]. If such an E can be found, invoke FindMin on the subtree whose root node is referenced by E.`ptr`. If no such entry is found, return NULL.

FM2: [Search leaf node] If R is a leaf, find the first entry E on R such that Consistent(E, q), and return E. If no such entry exists, return NULL.

GiST: search on linearly ordered domains

Algorithm Next(R, q, E)

Input: GiST rooted at R, predicate q, current entry E

Output: next entry in linear order that satisfies q

Sketch: return next entry on the same level of the tree if it satisfies q. Else return NULL.

N1: [next on node] If E is not the rightmost entry on its node, and N is the next entry to the right of E in order, and Consistent(N, q), then return N. If $\neg$Consistent(N, q), return NULL.

N2: [next on neighboring node] If E is the rightmost entry on its node, let P be the next node to the right of R on the same level of the tree (this can be found via tree traversal, or via sideways pointers in the tree, when available [LY81].) If P is non-existent, return NULL. Otherwise, let N be the leftmost entry on P. If Consistent(N, q), then return N, else return NULL.

GiST: insertion

Algorithm Insert(R, E, l)

Input: GiST rooted at R, entry $E = (p, \texttt{ptr})$, and level l, where p is a predicate such that p holds for all tuples reachable from `ptr`.

Output: new GiST resulting from insert of E at level l.

Sketch: find where E should go, and add it there, splitting if necessary to make room.

I1. [invoke ChooseSubtree to find where E should go] Let L = ChooseSubtree(R, E, l)

I2. If there is room for E on L, install E on L (in order according to Compare, if IsOrdered.) Otherwise invoke Split(R, L, E).

I3. [propagate changes upward] AdjustKeys(R, L).

GiST: insertion

Algorithm ChooseSubtree(R, E, l)

Input: subtree rooted at R, entry $E = (p, \texttt{ptr})$, level l

Output: node at level l best suited to hold entry with characteristic predicate $E.p$

Sketch: Recursively descend tree minimizing Penalty

CS1. If R is at level l, return R;

CS2. Else among all entries $F = (q, \texttt{ptr}')$ on R find the one such that Penalty(F, E) is minimal. Return ChooseSubtree($F.\texttt{ptr}', E, l$).

GiST: insertion

Algorithm Split(R, N, E)

Input: GiST R with node N, and a new entry $E = (p, \texttt{ptr})$.

Output: the GiST with N split in two and E inserted.

Sketch: split keys of N along with E into two groups according to PickSplit. Put one group onto a new node, and Insert the new node into the parent of N.

SP1: Invoke PickSplit on the union of the elements of N and $\{E\}$, put one of the two partitions on node N, and put the remaining partition on a new node N'.

SP2: [Insert entry for N' in parent] Let $E_{N'} = (q, \texttt{ptr}')$, where q is the Union of all entries on N', and $\texttt{ptr}'$ is a pointer to N'. If there is room for $E_{N'}$ on Parent(N), install $E_{N'}$ on Parent(N) (in order if IsOrdered.) Otherwise invoke Split(R, Parent(N), $E_{N'}$)[2].

SP3: Modify the entry F which points to N, so that $F.p$ is the Union of all entries on N.

GiST: insertion

Algorithm AdjustKeys(R, N)

Input: GiST rooted at R, tree node N

Output: the GiST with ancestors of N containing correct and specific keys

Sketch: ascend parents from N in the tree, making the predicates be accurate characterizations of the subtrees. Stop after root, or when a predicate is found that is already accurate.

PR1: If N is the root, or the entry which points to N has an already-accurate representation of the Union of the entries on N, then return.

PR2: Otherwise, modify the entry E which points to N so that $E.p$ is the Union of all entries on N. Then AdjustKeys(R, Parent(N).)

GiST: deletion

Deletion (1) maintains balance of tree, and (2) keeps keys as specific as possible.

Follows B+tree style on linearly ordered domains, and otherwise R-tree style.

GiST

Generalizes a wide variety of indexing proposals.

GiST

Generalizes a wide variety of indexing proposals.

Permits DB engine designers to focus on core extensible infrastructure.

Promotes deeper study of indexing

Wrap up

- Indexing, part 1: ordered indexes
  - B+ trees
  - R trees
  - GiST: generalized search trees

Wrap up

- Indexing, part 1: ordered indexes
  - B+ trees
  - R trees
  - GiST: generalized search trees
- next time: Indexing, part 2: hash indexes and indexability

Credits

- Our textbook (Silberschatz *et al.*, 2011)
- Ramakrishnan & Gehrke, 2003
- Hellerstein *et al.*, 1995