Indexing: Part II

CPS 216

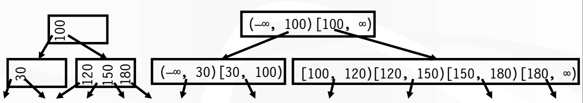
Advanced Database Systems

Announcements (February 5)

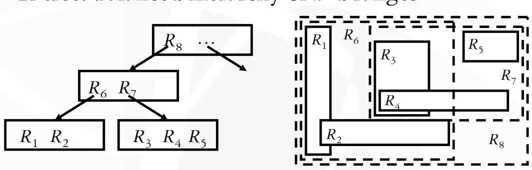
- Reading assignment for next week
  - "The" Google paper (due next Monday)
  - "The" query processing survey paper (due following Monday)
- Next recitation session tentatively scheduled for next Friday
- Midterm and course project proposal in four weeks

R-trees

- B-tree: balanced hierarchy of 1-d ranges

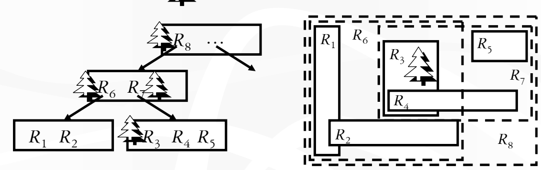


- R-tree: balanced hierarchy of *n*-d ranges



R-tree lookup

- Where am I?

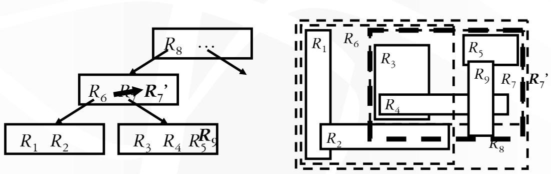


- Problem: search may go down many paths
  - Because regions may overlap
  - No performance guarantee like B-tree

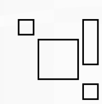
R-tree insertion

Insert R_9 into R-tree

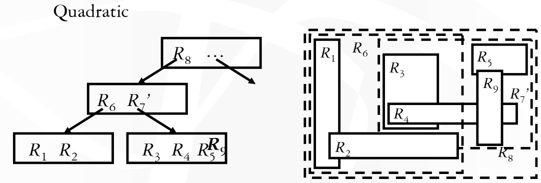
- Start from the root
- Pick a region containing R_9 and follow the child pointer
  - If none contains R_9, pick one and grow it to contain R_9
  - Pick the one that requires the least enlargement (why?)



R-tree insertion: split

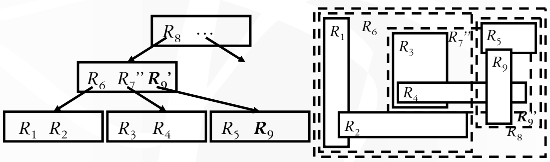


- If a node is too full, split
- Try to minimize the total area of bounding boxes
  - Exhaustive: try all possible splits
  - Quadratic: "seed" with the most wasteful pair; iteratively assign regions with strongest "preference"
  - Linear: "seed" with distant regions; iteratively assign others as Quadratic



R-tree insertion: split (cont'd)

❖ Split could propagate all the way up to the root (not shown in this example)



R*-tree

❖ R-tree
- Always tries to minimize the area of bounding boxes
- Quadratic splitting algorithm encourages small seeds and possibly long and narrow bounding boxes

❖ R*-tree (Beckmann et al., *SIGMOD* 1990)
- Consider other criteria, e.g.
  - Minimize overlap between bounding boxes
  - Minimize the margin (perimeter length) of a bounding box
- Forced reinserts
  - When a node overflows, reinsert "outer" entries
  - They may be picked up by other nodes, thus saving a split

R^+-tree

❖ Problem with R-tree
- Regions may overlap
- Search may go down many paths

❖ R^+-tree (Sellis et al., *VLDB* 1987)
- Regions in non-leaf nodes do not overlap
- Search only goes down one path
- Duplicate items in leaves
- But an insertion must now go down many paths!
  - R must be inserted into all R^+-tree leaves whose bounding boxes overlap with R
- A bigger tree

Review

❖ Tree-structured indexes
- ISAM
- B-tree and variants
- R-tree and variants
- Can we generalize? GiST!

Indexing user-defined data types

❖ Specialized indexes (ABCDEFG trees…)
- Redundant code: most trees are very similar
- Concurrency control and recovery especially tricky to get right

❖ Extensible B-trees and R-trees
- Examples: B-trees in Berkeley DB, B- and R-trees in Informix
- User-defined `compare()` function

☞ GiST (Generalized Search Trees)
- General (covers B-trees, R-trees, etc.)
- Easy to extend
- Built-in concurrency control and recovery

Structure of GiST

Balanced tree of $\langle p, ptr \rangle$ pairs

❖ p is a key predicate that holds for all objects found below ptr

❖ Every node has between kM and M index entries…
- k must be no more than ½ (why?)

❖ Except root, which only needs at least two children

❖ All leaves are on the same level

☞ User only needs to define what key predicates are

Defining key predicates

- boolean *Consistent*(entry *entry*, predicate *query*)
  - Return true if an object satisfying *query* might be found under *entry*
- predicate *Union*(set<entry> *entries*)
  - Return a predicate that holds for all objects found under *entries*
- real *Penalty*(entry *entry*1, entry *entry*2)
  - Return a penalty for inserting *entry*2 into the subtree rooted at *entry*1
- (set<entry>, set<entry>) *PickSplit*(set<entry> *entries*)
  - Given $M+1$ entries, split it into two sets, each of size at least kM

Index operations

- Search
  - Just follow pointer whenever *Consistent*( ) is true
- Insert
  - Descent tree along least increase in *Penalty*( )
  - If there is room in leaf, insert there; otherwise split according to *PickSplit*( )
  - Propagate changes up using *Union*( )
- Delete
  - Search for entry and delete it
  - Propagate changes up using *Union*( )
  - On underflow
    - If keys are ordered, can borrow/coalesce in B-tree style
    - Otherwise, reinsert stuff in the node and delete the node

GiST over R (B^+-tree)

- Logically, keys represent ranges $[x, y)$
- Query: find keys that overlap with $[a, b)$
- *Consistent*(*entry*, $[a, b)$): say *entry* has key $[x, y)$
  - $x < b$ and $y > a$, i.e., overlap
- *Union*(*entries*): say *entries* $= \{[x_i, y_i)\}$
  - $[\min(\{x_i\}), \max(\{y_i\}))$
- *Penalty*($entry_1$, $entry_2$): say they have keys $[x_1, y_1)$ and $[x_2, y_2)$
  - $\max(y_2 - y_1, 0) + \max(x_1 - x_2, 0)$, except boundary cases
- *PickSplit*(*entries*)
  - Sort *entries* and split evenly
- Plus a special *Compare*(*entry*, *entry*) for ordered keys

Key compression

- Without compression, GiST would need to store a range instead of a single key value in order to support B^+-tree
- Two extra methods: *Compress*/*Decompress*
- For B^+-tree
  - *Compress*(*entry*): say *entry* has key $[x, y)$
    - x, assuming next entry starts with y, except boundary cases
  - *Decompress*($\langle x, ptr \rangle$)
    - $[x, y)$, assuming next entry starts with y, except boundary cases
  - ☞This compression is lossless: *Decompress*(*Compress*(e)) $= e$

GiST over R^2 (R-tree)

- Logically, keys represent bounding boxes
- Query: find stuff that overlaps with a given box

  Abusing notation a bit below…
- *Consistent*(*key_box*, *query_box*)
  - *key_box* overlaps with *query_box*
- *Union*(*boxes*)
  - Minimum bounding box of *boxes*
- *Penalty*(box_1, box_2)
  - Area of *Union*($\{box_1, box_2\}$) – area of box_1
- *PickSplit*(*boxes*)
  - R-tree algorithms (e.g., minimize total area of bounding boxes)
- *Compare*(*box*, *box*)?

GiST over $P(Z)$ (RD-tree)

- Logically, keys represent sets
- Queries: find all sets that intersect with a given set
- *Consistent*(*key_set*, *query_set*)
  - *key_set* intersects with *query_set*
- *Union*(*sets*)
  - Union of *sets*
- *Penalty*(set_1, set_2)
  - $|$ *Union*($\{set_1, set_2\}$) $| - | set_1 |$
- *PickSplit*(*sets*)
  - Much like R-tree (e.g., minimize total cardinality)
- *Compare*(*set*, *set*)?
- *Compress*/*Decompress*: bloomfilters, rangesets, etc.
  - *Decompress*(*Compress*(*set*)) ? *set*
  - Lossy: *Decompress*(*Compress*(*set*)) $\supseteq$ *set*

Next

- ❖ Hash-based indexing
- ❖ Text indexing