R-tree: Indexing Structure for Data in Multi-dimensional Space

Feifei Li

(Many slides made available by Ke Yi)
Until now: Data Structures

- General planar range searching (in 2-dimensional space):
  - kdB-tree: $O(\sqrt{N/B} + T/B)$ query, $O(N/B)$ space
Other results

• Many other results for e.g.
  – Higher dimensional range searching
  – Range counting, range/stabbing max, and stabbing queries
  – Halfspace (and other special cases) of range searching
  – Queries on moving objects
  – Proximity queries (closest pair, nearest neighbor, point location)
  – Structures for objects other than points (bounding rectangles)

• Many heuristic structures in database community
Point Enclosure Queries

• Dual of planar range searching problem
  – Report all rectangles containing query point \((x,y)\)

• Internal memory:
  – Can be solved in \(O(N)\) space and \(O(\log N + T)\) time
  – Persistent interval tree
Rectangle Range Searching

- Report all rectangles intersecting query rectangle $Q$

- Often used in practice when handling complex geometric objects
  - Store minimal bounding rectangles (MBR)
**Rectangle Data Structures: R-Tree** [Guttman, SIGMOD84]

- Most common practically used rectangle range searching structure
- Similar to B-tree
  - Rectangles in leaves (on same level)
  - Internal nodes contain MBR of rectangles below each child
- Note: Arbitrary order in leaves/grouping order
Example
Example
Example
Example
• (Point) Query:
  – Recursively visit relevant nodes
Query Efficiency

• The fewer rectangles intersected the better
Rectangle Order

• Intuitively
  – Objects close together in same leaves
    \(\Rightarrow\) small rectangles \(\Rightarrow\) queries descend in few subtrees

• Grouping in internal nodes?
  – Small area of MBRs
  – Small perimeter of MBRs
  – Little overlap among MBRs
R-tree Insertion Algorithm

• When not yet at a leaf (*choose subtree*):
  – Determine rectangle whose area increment after insertion is smallest (small area heuristic)
  – Increase this rectangle if necessary and recurse

• At a leaf:
  – Insert if room, otherwise *Split Node* (while trying to minimize area)
Node Split

New MBRs
Linear Split Heuristic

• Determine the furthest pair $R_1$ and $R_2$ : the seeds for sets $S_1$ and $S_2$
• While not all MBRs distributed
  – Add next MBR to the set whose MBR increases the least
Quadratic Split Heuristic

• Determine R1 and R2 with largest area(MBR of R1 and R2)-area(R1) - area(R2): the seeds for sets S1 and S2
• While not all MBRs distributed
  – Determine of every not yet distributed rectangle R_j:
    d_1 = area increment of S_1 ∪ R_j
    d_2 = area increment of S_2 ∪ R_j
  – Choose R_i with maximal |d_1-d_2| and add to the set with smallest area increment
R-tree Deletion Algorithm

- Find the leaf (node) and delete object; determine new (possibly smaller) MBR
- If the node is too empty:
  - Delete the node recursively at its parent
  - Insert all entries of the deleted node into the R-tree
R*-trees [Beckmann et al. SIGMOD90]

- Why try to minimize area?
  - Why not overlap, perimeter,…

- R*-tree:
  - Better heuristics for
    Choose Subtree and Split Node
R-Tree Variants

• Many, many R-tree variants (heuristics) have been proposed

• Often bulk-loaded R-trees are used
  – Much faster than repeated insertions
  – Better space utilization
  – Can optimize more “globally”
  – Can be updated using previous update algorithms
How to Build an R-Tree

• Repeated insertions
  – [Guttman84]
  – R\textsuperscript{+}-tree [Sellis et al. 87]
  – R\textsuperscript{*}-tree [Beckmann et al. 90]

• Bulkloading
  – Hilbert R-Tree [Kamel and Faloutos 94]
  – Top-down Greedy Split [Garcia et al. 98]
  – Advantages:
    * Much faster than repeated insertions
    * Better space utilization
    * Usually produce R-trees with higher quality
R-Tree Variant: Hilbert R-Tree

- To build a Hilbert R-Tree (cost: $O(N/B \log_{M/B} N)$ I/Os)
  - Sort the rectangles by the Hilbert values of their centers
  - Build a B-tree on top
Z-ordering

• Basic assumption: Finite precision in the representation of each co-ordinate, \( K \) bits \( (2^K \) values) 
• The address space is a square (image) and represented as a \( 2^K \times 2^K \) array 
• Each element is called a pixel
Z-ordering

• Impose a linear ordering on the pixels of the image → 1 dimensional problem

\[ Z_A = \text{shuffle}(x_A, y_A) = \text{shuffle}(\text{"01"}, \text{"11"}) \]
\[ = 0111 = (7)_{10} \]
\[ Z_B = \text{shuffle}(\text{"01"}, \text{"01"}) = 0011 \]
Z-ordering

- Given a point \((x, y)\) and the precision \(K\) find the pixel for the point and then compute the z-value
- Given a set of points, use a B+-tree to index the z-values
- A range (rectangular) query in 2-d is mapped to a set of ranges in 1-d
Queries

• Find the z-values that contained in the query and then the ranges

\[ Q_A \rightarrow \text{range [4, 7]} \]

\[ Q_B \rightarrow \text{ranges [2,3] and [8,9]} \]
Handling Regions

• A region breaks into one or more pieces, each one with different z-value
• We try to minimize the number of pieces in the representation: precision/space overhead trade-off

\[ Z_{R1} = 0010 = (2) \]
\[ Z_{R2} = 1000 = (8) \]
\[ Z_G = 11 \]

("11" is the common prefix)
Z-ordering for Regions

- Break the space into 4 equal quadrants: level-1 blocks
- Level-i block: one of the four equal quadrants of a level-(i-1) block
- Pixel: level-K blocks, image level-0 block
- For a level-i block: all its pixels have the same prefix up to $2i$ bits; the z-value of the block
Hilbert Curve

- We want points that are close in 2d to be close in the 1d
- Note that in 2d there are 4 neighbors for each point where in 1d only 2.
- Z-curve has some “jumps” that we would like to avoid
- Hilbert curve avoids the jumps: recursive definition
It has been shown that in general Hilbert is better than the other space filling curves for retrieval [Jag90]

Hi (order-i) Hilbert curve for $2^i \times 2^i$ array
• A: plane-sweep on HILBERT curve!
R-trees - variations

- A: plane-sweep on HILBERT curve!
- In fact, it can be made dynamic (how?), as well as to handle regions (how?)
R-trees - variations

- Dynamic (‘Hilbert R-tree):
  - each point has an ‘h’-value (hilbert value)
  - insertions: like a B-tree on the h-value
  - but also store MBR, for searches
R-trees - variations

• Data structure of a node?

~B-tree

LHV  x-low, y-low
x-high, y-high
ptr

h-value >= LHV & MBRs: inside parent MBR
R-trees - variations

• Data structure of a node?

~ R-tree

- LHV: x-low, y-low, x-high, y-high
- ptr
- h-value >= LHV & MBRs: inside parent MBR
Theoretical Musings

• None of existing R-tree variants has worst-case query performance guarantee!
  – In the worst-case, a query can visit all nodes in the tree even when the output size is zero

• R-tree is a generalized kdB-tree, so can we achieve $O(\sqrt{N/B} + T/B)$?

• Priority R-Tree [Arge, de Berg, Haverkort, and Yi, SIGMOD04]
  – The first R-tree variant that answers a query by visiting $O(\sqrt{N/B} + T/B)$ nodes in the worst case
    * $T$: Output size
  – It is optimal!
    * Follows from the kdB-tree lower bound.
Roadmap

• Pseudo-PR-Tree
  – Has the desired $O(\sqrt{N/B} + T/B)$ worst-case guarantee
  – Not a real R-tree

• Transform a pseudo-PR-Tree into a PR-tree
  – A real R-tree
  – Maintain the worst-case guarantee

• Experiments
  – PR-tree
  – Hilbert R-tree (2D and 4D)
  – TGS-R-tree
Pseudo-PR-Tree

1. Place $B$ extreme rectangles from each direction in priority leaves
2. Split remaining rectangles by $x_{\text{min}}$ coordinates (round-robin using $x_{\text{min}}$, $y_{\text{min}}$, $x_{\text{max}}$, $y_{\text{max}}$ – like a 4d kd-tree)
3. Recursively build sub-trees

Query in $O(\sqrt{\frac{N}{B}} + \frac{T}{B})$ I/Os
- $O(T/B)$ nodes with priority leaf completely reported
- $O(\sqrt{\frac{N}{B}})$ nodes with no priority leaf completely reported
Pseudo-PR-Tree: Query Complexity

- Nodes $v$ visited where all rectangles in at least one of the priority leaves of $v$’s parent are reported: $O(T/B)$
- Let $v$ be a node visited but none of the priority leaves at its parent are reported completely, consider $v$’s parent $u$

$x_{\text{max}} = x_{\text{min}}(Q)$

$y_{\text{min}} = y_{\text{max}}(Q)$
Pseudo-PR-Tree: Query Complexity

• The cell in the 4d kd-tree of $u$ is intersected by two different 3-dimensional hyper-planes defined by sides of query $Q$
• The intersection of each pair of such 3-dimensional hyper-planes is a 2-dimensional hyper-plane
• Lemma: # of cells in a $d$-dimensional kd-tree that intersect an axis-parallel $f$-dimensional hyper-plane is $O((N/B)^{f/d})$
• So, # such cells in a 4d kd-tree: $O(\sqrt{N / B})$
• Total # nodes visited: $O(\sqrt{N / B} + T / B)$
PR-tree from Pseudo-PR-Tree
Query Complexity Remains Unchanged

\[ \sqrt{\frac{N}{B^3}} + \sqrt{\frac{N}{B^2}} / B + \sqrt{\frac{N}{B}} / B^2 + T / B^3 \]

Next level:

\[ \sqrt{\frac{N}{B^2}} + \sqrt{\frac{N}{B}} / B + T / B^2 \]

# nodes visited on leaf level \( \sqrt{\frac{N}{B}} + T / B \)
PR-Tree

- PR-tree construction in $O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os
  - Pseudo-PR-tree in $O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os
  - Cost dominated by leaf level

- Updates
  - $O(\log_B N)$ I/Os using known heuristics
    * Loss of worst-case query guarantee
  - $O(\log^2_B N)$ I/Os using logarithmic method
    * Worst-case query efficiency maintained

- Extending to $d$-dimensions
  - Optimal $O((N/B)^{1-1/d} + T/B)$ query