R-Trees for Spatial Indexing

Yanlei Diao
UMass Amherst
Feb 27, 2007

Spatial Indexing

- Applications: when we need to deal with n dimensional objects
  - Geographic Information Systems (GIS)
  - Very Large Scale Integrated Circuit (VLSI)
- Query workload: equality and range predicates in the n dimensional space
  - Find points in a polygon
  - Find polygons in a polygon
  - Find polygons overlapping with a polygon
  - Find polygons containing a polygon

R-tree: A Spatial Index

- Like B-tree
  - Height-balanced given arbitrary inserts/deletes.
  - Minimum 50% occupancy (except for root).
- Unlike B-tree
  - In a leaf node, data entry has the form <minimum bounding rectangle MBR, record id rid>
  - In a non-leaf node, index entry has the form <minimum bounding rectangle MBR, child node pointer p>
  - No particular order among data entries at the leaf level.
  - Why rectangles? Why not use a more precise description?
Search

- Given a $n$-dimensional search rectangle $S$, find all data entries that overlap with $S$.
  - At each non-leaf node, find all MBRs overlapping with $S$, follow their child pointers.
  - At each leaf node, find all MBRs overlapping with $S$, retrieve the corresponding data records.
  - For each data record, do a detailed comparison.
- Need to traverse multiple paths! (DFS)

Insert

- Insert a data entry $E <\text{MBR}, \text{rid}>$:
  - Step 1: Invoke $\text{ChooseSubtree}$ to descend from the root and select a leaf node $L$ for insertion.
  - Step 2: Insert $E$ to $L$.
    - If $L$ has room, insert $E$.
    - Otherwise, invoke $\text{SplitNode}$ to split entries in $L$ and $E$ into the $L$ node and a new LL node.
  - Step 3: Invoke $\text{AdjustTree}$ to propagate changes upward, passing $L$ and LL if split.
    - Adjust covering rectangles along the path bottom-up.
    - Propagate node splits as necessary.
**ChooseSubtree**

- **ChooseSubtree**: Descend from the root to select a leaf node for placing a new data entry E.
  - At each level of the tree, pick an index (data) entry using the least enlargement of the MBR. Ties broken by picking a MBR with the smallest area.
  - Heuristic-based algorithm. Other alternatives exist.
  - More on this when discussing R*-tree.

**SplitNode**

- **SplitNode**: Split M+1 entries (M from the node L, and the new entry E) between L and a new node LL.
  - **Exhaustive**: enumerate all legal subsets of all M+1 entries, considering the minimum occupancy requirement.
  - Guarantees to find the optimal split, but at an exponential cost!

**SplitNode (Contd.)**

- **SplitNode**: Split entries in the node L plus the new entry E between L and a new node LL.
  - **Quadratic**: sacrifice optimality for performance.
  - **Pick seeds**: Compute the area increase if two entries (Ei, Ej) are merged. Pick the pair with largest area increase, treat as the seeds for two groups.
  - **Split**: Greedily add other entries to the two groups by
    - picking the next E' that creates the max. difference in area increase between the two groups,
    - adding E' to the group with less area increase (ties resolved by picking the group with fewer entries).
SplitNode (Contd.)

- **SplitNode:** Split entries in the node \( L \) plus the new entry \( E \) between \( L \) and a new node \( LL \).
  - **Linear:** Like quadratic, but pick seeds differently.
    - For each dimension, pick the entry with lowest value and the other with the highest value, take the distance, normalize with the length of the dimension.
    - Among all, pick the dimension with highest normalized distance.
    - Treat the two extremes along that dimension as the seeds.
  - In practice, some systems (e.g. Postgres) seem to use the 50% minimum occupancy and quadratic split.

Delete

- **Remove a data entry \( E < MBR, rid > \):**
  - **Step 1:** Invoke \( \text{FindLeaf} \) to descend from the root and locate a leaf node \( L \) containing \( E \).
    - Like search may traverse multiple paths, but would expect just one match.
  - **Step 2:** Delete \( E \) from \( L \). Invoke \( \text{CondenseTree} \), passing \( L \), to adjust the tree.

CondenseTree

- **Start with a leaf node \( L \) with a data entry deleted.**
  - **Step 1:** Bottom-up adjustment. At each level, do:
    - If \( L \) has too few entries, delete \( L \) and its index entry in the parent node \( P \), add its entries \(<\text{entry}, \text{level}>\) to \( Q \) for reinsertions.
    - Otherwise, adjust its covering rectangle in \( P \).
    - Set \( L := P \), and repeat.
  - **Step 2:** Insert orphaned entries in \( Q \).
    - Use the Insert algorithm, but entries (including non-leaf ones) should be **inserted at the right level**.
  - Reinsertion incrementally refines the tree structure!
**R*-tree**

- Parameters of retrieval performance:
  - Area covered by a MBR, precisely, the area covered by a MBR but not its enclosed MBRs should be minimized.
  - Overlap between MBRs should be minimized.
  - Margin of a MBR, i.e., the sum of the lengths of edges of the MBR, should be minimized.
  - Benefit queries with quadratic rectangles.
  - Improve the structure.
  - Storage utility should be optimized.

**Issues with R-tree**

- Issues with R-tree:
  - SplitNode is designed to minimize the covering rectangles of the two nodes after split.
  - Can cause problems regarding other parameters.

**Insert in R*-tree**

- ChooseSubtree: Minimize different badness metrics depending on tree level:
  - Area cost at interior nodes, overlap cost at leaf nodes.
  - Why see?
- Interesting idea, but performance is only slightly better than R-tree.
- Can have some improvement in CPU cost, but benefit is not demonstrated.
Insert in R*-tree (Contd.)

- **SplitNode**: $O(NM\log M)$, more effective
  - For each dimension, sort entries by the lower value and then by upper value of their rectangles, $O(M\log M)$
  - For each sort, create $M-2m+2$ distributions of entries
    - $k$-th distribution: Group 1 $(m-1)+k$ entries, Group 2 contains the remaining entries.
    - Each group having $[m, M-m+1]$ entries implies $k \leq M-2m+2$.
  - For each distribution, compute goodness value based on (a) area-value, (b) margin-value, (c) overlap-value.
  - Select the best axis, choose the best distribution.
  - The algorithm here uses (b) to choose split axis, (c) to pick a distribution along that axis and (a) to break ties.

Forced Reinserts

- Structure varies with order of insertion.
  - Structure determined by earlier inserts is not suitable for good retrieval performance at the current situation.
  - Splits only cause local reorganization.
- So, force entries to be reinserted.
  - Used in R-tree to deal with deletion.
  - Used in R*-tree to handle entries in node splitting.
- Overflow treatment replaces node splitting.
  - Can reinsert both leaf and interior nodes.
  - For each entry to be reinserted, do it only if the call of OverflowTreatment is the first at a specific level.
  - Why? Reinsertion can go to the same node that needs to be treated. No need to reinsert any more in that case. Split instead.

Comments on Forced Reinserts

- Benefits:
  - decreases overlap of siblings
  - improves storage utilization
  - splits less often: causes better utilization?
  - shapes tend to be more quadratic: pack better, generate smaller parents...
- Experimental results show the best performance with reinsertion of 30% most “extreme” entries.
Comments on R*-tree

- Popular, because it outperforms R-tree on search.
- Problems:
  - Concurrency problems because of reinsertion.
  - Heuristics based, no formal analysis. Benefits proven empirically, but may vary with data sets used.
  - Not clear how different heuristics/algorithms contribute to the results, need some sort of cost decomposition.