R-Trees and GiST

Patryk Mastela

University of Michigan

pmastela@umich.edu

September 19, 2011
**Range Query:** find objects in a given range (e.g. find all museums in New York City)

- No index: need to scan through all objects. Inefficient!
- B+-tree: clusters based only on one dimension. Inefficient!
R-Tree Structure

- Non-leaf nodes contain entries in the form of $(l, \text{child-pointer})$
  - $l$ is an $n$-dimensional rectangle
- Leaf nodes contain entries in the form of $(l, \text{tuple-identifier})$
- $M$ is the maximum of entries which is usually given and $m$ is the minimum of entries in one node
R-Tree Properties

1. Every leaf node contains between $m$ and $M$ index records unless it is the root; the root can have less entries than $m$.

2. For each index record in a leaf node, $I$ is the smallest rectangle that spatially contains the $n$-dimensional data object represented by the indicated tuple.

3. Every non-leaf node has between $m$ and $M$ children unless it is the root.

4. For each entry in a non-leaf node, $i$ is the smallest rectangle that spatially contains the rectangles in the child node.

5. The root node has at least two children unless it is a leaf.

6. All leaves appear on the same level. That means the tree is balanced.
R-Trees and GiST
Search Algorithm

$T$ is root node of an R-Tree, find all index records whose rectangles overlap a search rectangle $S$.

**S1** [Search Subtree] If $T$ no leaf then check each entry $E$, whether $E.I$ overlaps $S$. For all overlapping entries, start **Search** on the subtree whose root node is pointed to by $E.p$.

**S2** [Search leaf node] If $T$ is a leaf, then check each entry $E$ whether $E.I$ overlaps $S$. If so, $E$ is a suitable entry.
Search Algorithm

Example
**S1** [Search Subtree] If $T$ no leaf then check each entry $E$, weather $E.I$ overlaps $S$. For all overlapping entries, start **Search** on the subtree whose root node is pointed to by $E.p$. 

---

**Example**

```
  E1 | E2

  E3  E4  E5

  abc  def  fgh

  E6  E7

  ijk  lmn
```
S1 [Search Subtree] If $T$ no leaf then check each entry $E$, weather $E.I$ overlaps $S$. For all overlapping entries, start **Search** on the subtree whose root node is pointed to by $E.p$. 

Example

```
  E1  E2  
 /     \
E3    E4  E5
  /     |    |
 a b c  d e  f g h

E6    E7
  /     |    |
i j k  l m  ```
S2 [Search leaf node] If $T$ is a leaf, then check each entry $E$ whether $E.I$ overlaps $S$. If so, $E$ is a suitable entry.
**S2** [Search leaf node] If $T$ is a leaf, then check each entry $E$ whether $E.I$ overlaps $S$. If so, $E$ is a suitable entry.

**Example**

![Diagram](image-url)
**S2** [Search leaf node] If $T$ is a leaf, then check each entry $E$ whether $E.I$ overlaps $S$. If so, $E$ is a suitable entry.

---

**Example**

```
   E1  E2
  /    \
E3  E4  E5
   /     \
  a  b  c  d  e  f  g  h
          /  \  /
         i  j  k  l  m
```
S2  [Search leaf node] If $T$ is a leaf, then check each entry $E$ whether $E.I$ overlaps $S$. If so, $E$ is a suitable entry.
S2 [Search leaf node] If \( T \) is a leaf, then check each entry \( E \) whether \( E.I \) overlaps \( S \). If so, \( E \) is a suitable entry.
S2 [Search leaf node] If $T$ is a leaf, then check each entry $E$ whether $E.I$ overlaps $S$. If so, $E$ is a suitable entry.

Example
S1  [Search Subtree] If \( T \) no leaf then check each entry \( E \), weather \( E.I \) overlaps \( S \). For all overlapping entries, start **Search** on the subtree whose root node is pointed to by \( E.p \).
**S2** [Search leaf node] If $T$ is a leaf, then check each entry $E$ whether $E.I$ overlaps $S$. If so, $E$ is a suitable entry.
S2  [Search leaf node] If $T$ is a leaf, then check each entry $E$ whether $E.I$ overlaps $S$. If so, $E$ is a suitable entry.
**S2** [Search leaf node] If $T$ is a leaf, then check each entry $E$ whether $E.I$ overlaps $S$. If so, $E$ is a suitable entry.

**Example**

![Diagram showing the search algorithm with nodes E1, E2, E3, E4, E5, E6, E7, and overlapping regions for each node.](image-url)
New entry $E$ will be inserted into a given R-Tree.

1. [Find position for new record] Start \texttt{ChooseLeaf} to select a leaf node $L$ in which to place $E$.

2. [Add record to leaf node] If node $L$ has enough space for a new entry then add $E$. Else start \texttt{SplitNode} to obtain $L$ and $LL$ containing $E$ and all the old entries of $L$.

3. [Propagate changes upward] Start \texttt{AdjustTree} on node $L$ and if a split was performed then also passing $LL$.

4. [Grow tree taller] If node split propagation caused the root to split, create a new root whose children are the two resulting nodes.
ChooseLeaf selects a leaf node to place a new entry $E$.

**CL1** [Initialize] Set $N$ to be the root node.

**CL2** [Leaf check] If $N$ is a leaf then return $N$.

**CL3** [Choose subtree] If $N$ is not a leaf node then let $F$ be the entry in $N$ whose MBR $F.I$ needs least enlargement to include $E.I$. When there are more qualify entries in $N$, the entry with the rectangle of the smallest area is chosen.

**CL4** [Descend until a leaf is reached] $N$ is set to the child node $F$ which is pointed to by $F.p$ and repeat from **CL2**
AdjustTree Algorithm

Leaf node $L$ is upwarded to the root while adjusting covering rectangles. If necessary it comes to propagating node splits.

**AT1** [Initialize] $N$ is equal $L$.

**AT2** [Check if done] stop if $N$ is the root

**AT3** [Adjust covering MBR in parent entry] $P$ is the parent node of $N$ and $E_N$ the entry of $N$ in $P$. $E_N.I$ is adjusted so that all rectangles in $N$ are tightly enclosed.

**AT4** [Propagate node split upward] If $N$ has a partner $NN$ which was split previously then create a new entry with $E_{NN}.p$ pointing to $NN$ and $E_{NN}.I$ enclosing all rectangles in $NN$. If there is room in $P$ then add $E_{NN}$. Otherwise start **SplitNode** to get $P$ and $PP$ which include $E_{NN}$ and all old entries of $P$.

**AT5** [Move up to next level] $N$ is equal $L$ and if a split occurred then $NN$ is equal $PP$. Repeat from **AT2**
Insert Algorithm

Example
Insert Algorithm

Example

E1 E2
E3 E4 E5
E6 E7

abc def gh

ijkl m
Insert Algorithm

Example

```
E1  E2
|     |
E3--E4--E5
  |     |
  a-b-c| d-e-f-g-h

E6--E7
  |     |
  i-j-k| l-m
```
Insert Algorithm

Example

```
E1  E2

E3  E4  E5

a   b   c   d   e   f   g   h

E6  E7

i   j   k   l   m
```
Insert Algorithm

Example

R-Trees and GiST
Delete entry $E$ from an R-Tree.

D1 [Find node containing record] Start $\text{FindLeaf}$ to find the leaf node $L$ containing $E$. If search unsuccessful then terminate.

D2 [Delete record] Remove $E$ from $L$.

D3 [Propagate record] Start $\text{CondenseTree}$ on $L$.

D4 [Shorten tree] If the root node has only one child after adjusting then make the child the new root.
FindLeaf Algorithm

Root node is $T$, the leaf node containing the index entry $E$ is to find.

**FL1** [Search subtree] If $T$ is not a leaf, then check each entry $F$ in $T$ to determine when $F.I$ overlaps $E.I$. For all these entries FindLeaf starts on the subtree whose root is pointed to by $F.p$ until $E$ is found or each entry has been checked.

**FL2** [Search leaf node for record] If $T$ is a leaf, then check each entry to see when it matches $E$. If $E$ is found, then return $T$. 
CondenseTree Algorithm I

Given is a leaf node \( L \) from which an entry has been deleted. If \( L \) has too few entries then eliminate it from the tree. After that, the remaining entries in \( L \) are reinserted in the tree. This procedure is repeated until the root. Also adjust all covering rectangles on the path to the root, making them smaller, if possible.

**CT1** [Initialize] \( N \) is equal \( L \). Initialize a list \( Q \) which consists of eliminate nodes as empty.

**CT2** [Find parent entry] If \( N \) is the root, then go to **CT6**. Else \( P \) is the parent node of \( N \), and \( E_N \) the entry of \( N \) in \( P \).

**CT3** [Eliminate underflow node] If \( N \) has fewer than \( m \) entries, then eliminate \( E_N \) from \( P \) and add \( N \) to list \( Q \).

**CT4** [Adjust covering rectangle] If \( N \) has not been deleted, then adjust \( E_N \).I to tightly contain all entries in \( N \).

**CT5** [Move up one level in tree] \( N \) is equal \( P \) and repeat from **CT2**.
CondenseTree Algorithm II

CT6 [Re-insert orphaned entries] Every entry in Q is inserted. Leaf nodes are inserted like in Insertion. However, entries from higher-level nodes must be placed higher in the tree, so that leaves of their dependent subtrees will be on the same level as leaves of the main tree.
Delete $l$ where $m = 2$ and $M = 3$.

Example

```
Delete Algorithm

Example

Delete $l$ where $m = 2$ and $M = 3$.

Example

```

Patryk Mastela

R-Trees and GiST
Delete Algorithm

Deleting left $E_7$ with an underflow.

Example

```
+----+----+
| E1 | E2 |
+----+----+
    /  \
   /    \
  +----+----+
  | E3 | E4 | E5 |
  +----+----+
      /  \
     /    \
    +----+----+
    | a   | b   | c   | d   | e   | f   | g   | h   |
    +----+----+
            /  \
            /    \
          +----+----+
          | E6   | E7   |
          +----+----+
                /  \
                /    \
              +----+----+
              | i     | j     | k     | m |
              +----+----+
```
Save $m$ and remove $E7$; $E2$ now has an underflow.

Example
Save $E_6$ and remove $E_2$. 

Example
The root has only one child and thus the child becomes the new root.

Example

```
+---+---+---+
| E3 | E4 | E5 |
+---+---+---+
    +---+---+---+
    | a | b | c |
    +---+---+---+
    +---+---+---+
    | d | e |
    +---+---+---+
    +---+---+---+
    | f | g | h |
    +---+---+---+
```
Delete Algorithm

Insert $E6$ which causes the root to split.

Example
Delete Algorithm

Insert \( l \); \( E6 \) is split.

Example
GiST Motivation

- New applications
  - Geographic information systems
  - Multimedia systems
  - CAD tools
  - Document libraries
  - Sequence databases
  - Fingerprint identification systems
  - Biochemical databases

- Rapid data type introduction; in need of equally adaptable search trees.
Prior Approaches

- **Specialized Search Trees**
  - e.g. Spatial Search Trees (R-Trees)
  - High cost of implementation and maintenance

- **Search Trees for Extensible Data Types**
  - e.g. B+-trees can be used to index any data with linear ordering
  - Extending data does not extend set of queries supported by tree
GiST: A New Hope

- A third direction for extending search tree technology
- Extensible in both the data types it can index and the queries it can support
- Allows new data types to be indexed in a manner that supports the queries natural to the types
- Unifies previously disparate structures used for currently common data types
  - e.g. B+-trees and R-trees can be implemented as extensions of the GiST. Single code base yet indexes multiple dissimilar applications.
- Balanced tree; high fanout
- **Search Key**: any arbitrary predicate that holds for each datum below the key
- **Search Tree**: hierarchy of categorizations, in which each categorization holds for all data stored under it in the hierarchy
Generalized Search Tree

- Balanced tree of variable fanout between $kM$ and $M$
  - $k$ is the minimum fill factor of the tree, $\frac{2}{M} \leq k \leq \frac{1}{2}$
  - The exception is the root which may have a fanout between 2 and $M$
- Non-leaf nodes $(p, ptr)$
  - $p$, predicate that is used as a search key
  - $ptr$, pointer to another tree node
- Leaf nodes $(p, ptr)$
  - $p$, predicate that is used as a search key
  - $ptr$, identifier of some tuple in the database
GiST Properties

1. Unless the node is root every node contains between $kM$ and $M$ index entries.
2. For leaf nodes, $p$ is true when instantiated with the values from the indicated tuple.
3. For non-leaf nodes, $p$ is true when instantiated with the values of any tuple reachable from $ptr$.
4. The root has at least two children unless it is a leaf.
5. All leaves appear on the same level.
Key Methods

- **Search**
  - **Consistent**\((E, q)\): Asks \(E \cdot p \land q\)

- **Characterization**
  - **Union**\((P)\): returns new predicate that holds for all tuples in \(P\)

- **Categorization**
  - **Penalty**\((E_1, E_2)\): penalty for inserting \(E_2\) into \(E_1\)
  - **PickSplit**\((P)\): split \(P\) into two group of entries

- **Compression**
  - **Compress**\((E)\): returns compressed representation of \(p\)
  - **Decompress**\((E)\): returns an entry such \((r, ptr)\) such that \(p \rightarrow r\)
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.
Find where $E$ should go, and add it there, splitting if necessary to make room.

1. [invoke ChooseSubtree to find where $E$ should go] Let $L = \text{ChooseSubtree}(R, E, l)$
2. If there is room for $E$ on $L$, install $E$ on $L$ (in order according to Compare, if IsOrdered.) Otherwise invoke Split($R, L, E$).
Remove $E$ from its leaf node. If this causes underflow, adjust tree accordingly. Update predicates in ancestors to keep them as specific as possible.

D1 [Find node containing entry] Invoke Search($R, E, p$) and find leaf node $L$ containing $E$. Stop if $E$ not found.

D2 [Delete entry.] Remove $E$ from $L$.

D3 [Propagate changes.] Invoke CondenseTree($R, L$).

D4 [Shorten tree.] If the root node has only one child after the tree has been adjusted, make the child the new root.
■ **Consistent**\((E, q)\) returns true if
  - \(q = \text{Contains([}x_q, y_q), v): (x_p < x_q \land y_q > x_p)\)
  - \(q = \text{Equal}(x_q, v): x_p \leq x_q < y_p\)

■ **Union**\((P)\) returns \([\text{MIN}(x_1, \ldots, x_n), \text{MAX}(y_1, \ldots, y_n)]\)

■ **Penalty**\((E, F)\)
  - If \(E\) is the leftmost pointer on its node, returns \(\text{MAX}(y_2 - y_1, 0)\)
  - If \(E\) is the rightmost pointer on its node, returns \(\text{MAX}(x_1 - x_2, 0)\)
  - Otherwise, returns \(\text{MAX}(y_2 - y_1, 0) + \text{MAX}(x_1 - x_2, 0)\)

■ **PickSplit**\((P)\) Let the first \(\left\lfloor \frac{|P|}{2} \right\rfloor\) entries in order go in the left group, and the rest in the right
Compress\((E)\) If \(E\) is the leftmost key on a non-leaf node return 0 bytes otherwise, returns \(E.p.x\)

Decompress\((E)\)
- If \(E\) is the leftmost key on a non-leaf node let \(x = -\infty\)
  otherwise let \(x = E.p.x\)
- If \(E\) is the rightmost key on a non-leaf node let \(y = \infty\). If \(E\) is other entry in a non-leaf node, let \(y\) = the value stored in the next key. Otherwise, let \(y = x + 1\)
GiSTs over R-Trees

Key: \((x_{ul}, y_{ul}, x_{lr}, y_{lr})\)

- Query predicates
  - Contains \(((x_{ul1}, y_{ul1}, x_{lr1}, y_{lr1}), (x_{ul2}, y_{ul2}, x_{lr2}, y_{lr2}))\)
    - Returns true if
      \((x_{ul1} \leq x_{ul2}) \land (y_{ul1} \geq y_{ul2}) \land (x_{lr1} \geq x_{lr2}) \land (y_{lr1} \leq y_{lr2})\)
  - Overlaps \(((x_{ul1}, y_{ul1}, x_{lr1}, y_{lr1}), (x_{ul2}, y_{ul2}, x_{lr2}, y_{lr2}))\)
    - Returns true if
      \((x_{ul1} \leq x_{lr2}) \land (y_{ul1} \geq y_{lr2}) \land (x_{ul2} \leq x_{lr1}) \land (y_{lr1} \leq y_{ul2})\)
  - Equal \(((x_{ul1}, y_{ul1}, x_{lr1}, y_{lr1}), (x_{ul2}, y_{ul2}, x_{lr2}, y_{lr2}))\)
    - Returns true if
      \((x_{ul1} = x_{ul2}) \land (y_{ul1} = y_{ul2}) \land (x_{lr1} = x_{lr2}) \land (y_{lr1} = y_{lr2})\)
GiSTs over R-Trees II

- **Consistent**($E, q$)
  - $p$ contains $(x_{ul1}, y_{ul1}, x_{lr1}, y_{lr1})$, and $q$ is either Contains, Overlaps, or Equals $(x_{ul2}, y_{ul2}, x_{lr2}, y_{lr2})$
  - Returns true if Overlaps $((x_{ul1}, y_{ul1}, x_{lr1}, y_{lr1}), (x_{ul2}, y_{ul2}, x_{lr2}, y_{lr2}))$

- **Union**($P$) returns coordinates of the maximum bounding rectangles of all rectangles in $P$.

- **Penalty**($E, F$) Compute $q = \text{Union}(E,F)$ and return $\text{area}(q) - \text{area}(E.p)$

- **PickSplit**($P$) Variety of algorithms are provided to best split the entries in a over-full node.

- **Compress**($E$) Form the bounding rectangle of $E.p$

- **Decompress**($E$) The identity function
Issues

- Performance
  - Data overlap
  - Lossy compression

- Implementation
  - Concurrency Control, Recovery and Consistency
  - Variable-Length Keys
  - Bulk Loading