CS618: Indexing and Searching Techniques in Databases
Hierarchical Structures

Arnab Bhattacharya
arnabb@cse.iitk.ac.in

Computer Science and Engineering,
Indian Institute of Technology, Kanpur
http://web.cse.iitk.ac.in/~cs618/

2nd semester, 2014-15

Mon 1200-1315, Tue 0900-1015 at CS101

B-tree

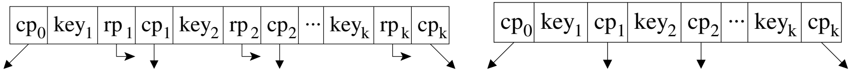
- Balanced hierarchical data structure
- Keys (and associated objects) are in secondary storage, i.e., disk
- A B-tree of order Θ has the following properties:
  1. Leaf nodes are in same level, i.e., the tree is balanced
  2. Root has at least 1 key
  3. Other internal nodes have between Θ and 2Θ keys
  4. An internal node with k keys have $k + 1$ children
  5. Child pointers in leaf nodes are null
- Branching factor is between $\Theta + 1$ and $2\Theta + 1$
- Pointer to the object corresponding to a key is stored alongside

B+-tree

- Most important variety of B-tree
- Internal nodes do *not* contain pointers to objects
- Often siblings are connected by pointers to avoid parent traversal

B+-tree

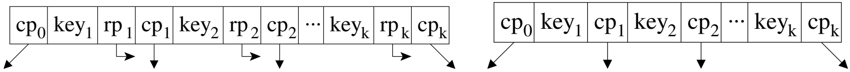
- Most important variety of B-tree
- Internal nodes do *not* contain pointers to objects
- Often siblings are connected by pointers to avoid parent traversal



B-tree node B+-tree node

B+-tree

- Most important variety of B-tree
- Internal nodes do *not* contain pointers to objects
- Often siblings are connected by pointers to avoid parent traversal



B-tree node B+-tree node

- More keys can fit in a B+-tree
- Height may be less

Order

- Order mainly determined by disk page size

Order

- Order mainly determined by disk page size
- Disk page capacity is C bytes
- Key consumes γ bytes
- Pointer to key (or object) requires η bytes

Order

- Order mainly determined by disk page size
- Disk page capacity is C bytes
- Key consumes γ bytes
- Pointer to key (or object) requires η bytes
- For B+-tree

Order

- Order mainly determined by disk page size
- Disk page capacity is C bytes
- Key consumes γ bytes
- Pointer to key (or object) requires η bytes
- For B+-tree

$$C \geq 2.\Theta.\gamma + (2.\Theta + 1).\eta$$
$$\Rightarrow \Theta_{\text{B+-tree}} = \left\lfloor \frac{C - \eta}{2(\gamma + \eta)} \right\rfloor$$

Order

- Order mainly determined by disk page size
- Disk page capacity is C bytes
- Key consumes γ bytes
- Pointer to key (or object) requires η bytes
- For B+-tree

$$C \geq 2.\Theta.\gamma + (2.\Theta + 1).\eta$$
$$\Rightarrow \Theta_{\text{B+-tree}} = \left\lfloor \frac{C - \eta}{2(\gamma + \eta)} \right\rfloor$$

- For B-tree

Order

- Order mainly determined by disk page size
- Disk page capacity is C bytes
- Key consumes γ bytes
- Pointer to key (or object) requires η bytes
- For B+-tree

$$C \geq 2.\Theta.\gamma + (2.\Theta + 1).\eta$$
$$\Rightarrow \Theta_{\text{B+-tree}} = \left\lfloor \frac{C - \eta}{2(\gamma + \eta)} \right\rfloor$$

- For B-tree

$$C \geq 2.\Theta.\gamma + 2.\Theta.\eta + (2.\Theta + 1).\eta$$
$$\Rightarrow \Theta_{\text{B-tree}} = \left\lfloor \frac{C - \eta}{2(\gamma + 2\eta)} \right\rfloor$$

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?
  - $\Theta_{\text{B+-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+4)} \right\rfloor = 170$

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?
  - $\Theta_{\text{B+-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+4)} \right\rfloor = 170$
  - $\Theta_{\text{B-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+2 \times 4)} \right\rfloor = 127$

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?
  - $\Theta_{\text{B+-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+4)} \right\rfloor = 170$
  - $\Theta_{\text{B-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+2 \times 4)} \right\rfloor = 127$
- What is the height of the B+-tree and the B-tree for 3×10^7 keys?

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?
  - $\Theta_{\text{B+-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+4)} \right\rfloor = 170$
  - $\Theta_{\text{B-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+2 \times 4)} \right\rfloor = 127$
- What is the height of the B+-tree and the B-tree for 3×10^7 keys?
  - Height of B+-tree is $\lceil \log_{2*170}(3 \times 10^7) \rceil = 3$

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?
  - $\Theta_{\text{B+-tree}} = \left\lfloor \frac{4\times1024-4}{2(8+4)} \right\rfloor = 170$
  - $\Theta_{\text{B-tree}} = \left\lfloor \frac{4\times1024-4}{2(8+2\times4)} \right\rfloor = 127$
- What is the height of the B+-tree and the B-tree for 3×10^7 keys?
  - Height of B+-tree is $\lceil \log_{2*170}(3 \times 10^7) \rceil = 3$
  - Height of B-tree is $\lceil \log_{2*127}(3 \times 10^7) \rceil = 4$

K-d-B tree

- Disk-based version of K-d-trees
- To combine multi-dimensional search efficiency of K-d-trees with disk I/O efficiency of B+-trees

K-d-B tree

- Disk-based version of K-d-trees
- To combine multi-dimensional search efficiency of K-d-trees with disk I/O efficiency of B+-trees
- Balanced
- Internal node contains region keys for indicating children
- Leaf nodes are called *point pages*
- Internal nodes are called *region pages*

K-d-B tree

- Disk-based version of K-d-trees
- To combine multi-dimensional search efficiency of K-d-trees with disk I/O efficiency of B+-trees
- Balanced
- Internal node contains region keys for indicating children
- Leaf nodes are called *point pages*
- Internal nodes are called *region pages*
- When leaf overflows, split leaf and create two leaves
- Adjust region keys accordingly

K-d-B tree

- Disk-based version of K-d-trees
- To combine multi-dimensional search efficiency of K-d-trees with disk I/O efficiency of B+-trees
- Balanced
- Internal node contains region keys for indicating children
- Leaf nodes are called *point pages*
- Internal nodes are called *region pages*
- When leaf overflows, split leaf and create two leaves
- Adjust region keys accordingly
- Region splits (hyperplanes) need not alternate

K-d-B tree

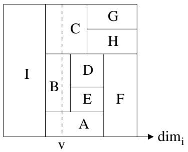
- Disk-based version of K-d-trees
- To combine multi-dimensional search efficiency of K-d-trees with disk I/O efficiency of B+-trees
- Balanced
- Internal node contains region keys for indicating children
- Leaf nodes are called *point pages*
- Internal nodes are called *region pages*
- When leaf overflows, split leaf and create two leaves
- Adjust region keys accordingly
- Region splits (hyperplanes) need not alternate
- Can underflow
- Space utilization can be very low

Splitting

- Split is always made using an axis-oriented hyperplane
- Hyperplane extends all the way along the range for the other dimensions
- Thus, children nodes may get split as well

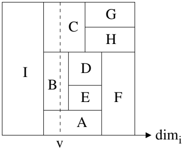
Splitting

- Split is always made using an axis-oriented hyperplane
- Hyperplane extends all the way along the range for the other dimensions
- Thus, children nodes may get split as well
- Region A is split at v
- B and C get split as well



Splitting

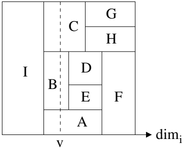
- Split is always made using an axis-oriented hyperplane
- Hyperplane extends all the way along the range for the other dimensions
- Thus, children nodes may get split as well
- Region *A* is split at *v*
- *B* and *C* get split as well



- Choice of split dimension
  - Dimension with largest range
  - Round robin

Splitting

- Split is always made using an axis-oriented hyperplane
- Hyperplane extends all the way along the range for the other dimensions
- Thus, children nodes may get split as well
- Region *A* is split at *v*
- *B* and *C* get split as well



- Choice of split dimension
  - Dimension with largest range
  - Round robin
- Choice of split value
  - Middle of range

Framework for hierarchical object-based structures

- n objects in a d-dimensional vector space
- Object represented by a hyper-dimensional *bounding* geometric *convex* "box"
- Minimum bounding boxes (rectangles or spheres or polygons) are organized in a *hierarchy*

Framework for hierarchical object-based structures

- n objects in a d-dimensional vector space
- Object represented by a hyper-dimensional *bounding* geometric *convex* "box"
- Minimum bounding boxes (rectangles or spheres or polygons) are organized in a *hierarchy*
- A larger box B_i at level i must *completely encompass* all smaller boxes B_{i-1} that lie in its subtree
- Balanced structure improves performance

Framework for hierarchical object-based structures

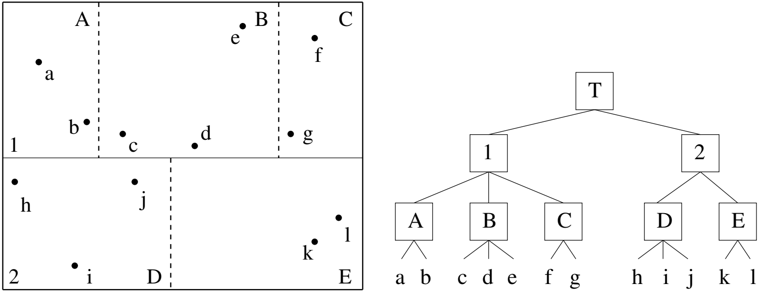
- n objects in a d-dimensional vector space
- Object represented by a hyper-dimensional *bounding* geometric *convex* "box"
- Minimum bounding boxes (rectangles or spheres or polygons) are organized in a *hierarchy*
- A larger box B_i at level i must *completely encompass* all smaller boxes B_{i-1} that lie in its subtree
- Balanced structure improves performance
- Root contains only one box corresponding to the entire space
- Assume that m boxes fit in a page
- Total number of leaves is $\lceil (n/m) \rceil$
- Total number of bounding boxes is $\lceil (n/m) \rceil + \lceil (n/m^2) \rceil + \cdots = O(n/m)$
- Size overhead of this index structure is, therefore, *linear*

Framework for hierarchical object-based structures

- n objects in a d-dimensional vector space
- Object represented by a hyper-dimensional *bounding* geometric *convex* "box"
- Minimum bounding boxes (rectangles or spheres or polygons) are organized in a *hierarchy*
- A larger box B_i at level i must *completely encompass* all smaller boxes B_{i-1} that lie in its subtree
- Balanced structure improves performance
- Root contains only one box corresponding to the entire space
- Assume that m boxes fit in a page
- Total number of leaves is $\lceil (n/m) \rceil$
- Total number of bounding boxes is $\lceil (n/m) \rceil + \lceil (n/m^2) \rceil + \cdots = O(n/m)$
- Size overhead of this index structure is, therefore, *linear*
- Height is $h = \lceil \log_m n \rceil$
- Thus, any object can be located in h disk accesses

Object pyramid

- This framework is called an object pyramid
- It provides *multi-resolution representation* of database objects

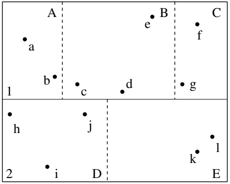


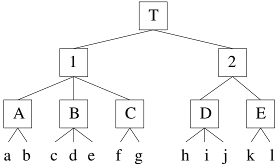
Point queries

- Query for q
  - Start with level 0
  - At level i, determine if q is in object D_i
    - If no, return
    - If yes, recursively search for all children in D_{i+1}

Point queries

- Query for q
  - Start with level 0
  - At level i, determine if q is in object D_i
    - If no, return
    - If yes, recursively search for all children in D_{i+1}
- Example: j
  - Start with T
  - Prune 1, proceed to 2 only
  - Then, only D and finally j
- May follow multiple paths when objects are non-disjoint
- May end up doing more work than sequential scan



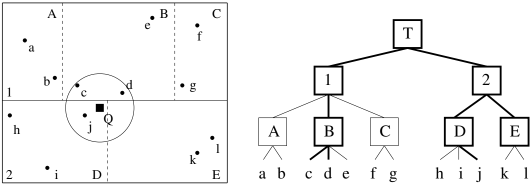


Range queries

- Query point q and a distance radius r
  - Traverse a child only if it intersects query "box"
  - Can be found by region intersection
  - Compute minimum and maximum distances from the query point to a "box" corresponding to a region

Range queries

- Query point q and a distance radius r
  - Traverse a child only if it intersects query "box"
  - Can be found by region intersection
  - Compute minimum and maximum distances from the query point to a "box" corresponding to a region
- Example
  - Both 1 and 2
  - Within 1, only B
  - Within 2, both D and E
  - Searching E is wasted but cannot be avoided
- *Window queries* are solved similarly

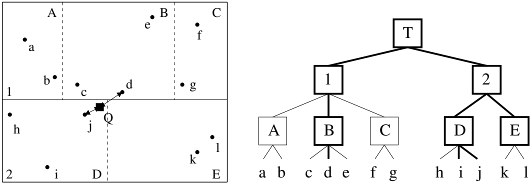


kNN queries

- Query point q and a positive integer k
  - Maintain max-heap of k current estimates
  - Range is dynamically maintained as k^{th} distance d_k
  - If minimum distance to region $> d_k$, then prune
  - Otherwise, insert into the max-heap

kNN queries

- Query point q and a positive integer k
  - Maintain max-heap of k current estimates
  - Range is dynamically maintained as k^{th} distance d_k
  - If minimum distance to region $> d_k$, then prune
  - Otherwise, insert into the max-heap
- Example
  - Assume, query first descends into 2 and then D and then j, i
  - Current kNN answer is then $\{j, i\}$ and $d_k = d(Q, i)$
  - E is traversed since minimum distance to E is less than d_k
  - No change in answer
  - Next, 1 and then B
  - Current kNN set is updated to $\{j, d\}$ with $d_k = d(Q, d)$
  - A and C are pruned since their minimum distances are greater than d_k



kNN searching order

- Efficiency depends heavily on the order in which children are searched

kNN searching order

- Efficiency depends heavily on the order in which children are searched
- Depth-first search
  - Descend to a leaf first
  - Get estimates of answer sets fast

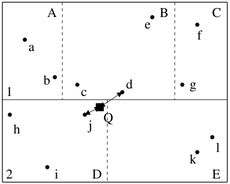
kNN searching order

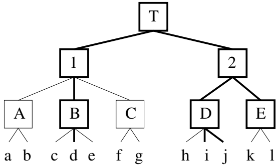
- Efficiency depends heavily on the order in which children are searched
- Depth-first search
  - Descend to a leaf first
  - Get estimates of answer sets fast
- Breadth-first search
  - Keep refining estimates level by level
  - Does not help as much

kNN searching order

- Efficiency depends heavily on the order in which children are searched
- Depth-first search
  - Descend to a leaf first
  - Get estimates of answer sets fast
- Breadth-first search
  - Keep refining estimates level by level
  - Does not help as much
- Best-first search
  - Maintains a priority queue (min-heap) of candidates
  - Examine the next *best* candidate from the min-heap
  - May traverse the nodes in any order

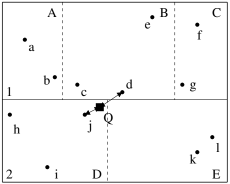
Example

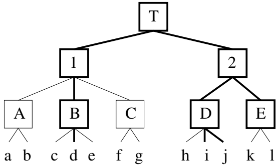




Extract	Insert	Priority queue	Answer	$d_{k=2}$

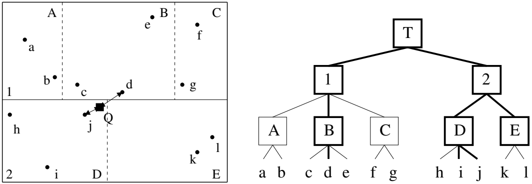
Example





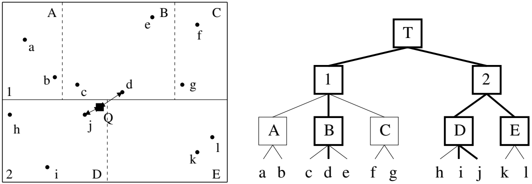
Extract	Insert	Priority queue	Answer	$d_{k=2}$
-	T	$(T, 0)$	Φ	∞

Example



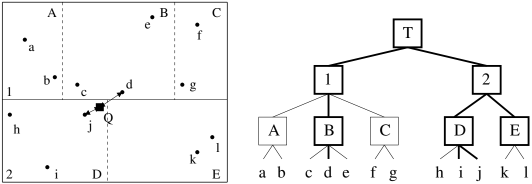
Extract	Insert	Priority queue	Answer	$d_{k=2}$
-	T	$(T, 0)$	Φ	∞
T	1, 2	$(2, 0)$, $(1, 4)$	Φ	∞

Example



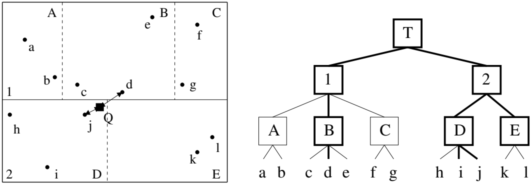
Extract	Insert	Priority queue	Answer	$d_{k=2}$
-	T	$(T, 0)$	Φ	∞
T	1, 2	$(2, 0)$, $(1, 4)$	Φ	∞
2	D, E	$(D, 0)$, $(E, 2)$, $(1, 4)$	Φ	∞

Example



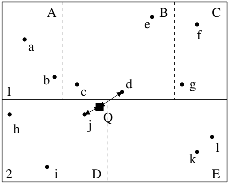
Extract	Insert	Priority queue	Answer	$d_{k=2}$
-	T	$(T, 0)$	Φ	∞
T	1, 2	$(2, 0)$, $(1, 4)$	Φ	∞
2	D, E	$(D, 0)$, $(E, 2)$, $(1, 4)$	Φ	∞
D	h, i, j	$(E, 2)$, $(1, 4)$, $(j, 6)$, $(h, 27)$, $(i, 30)$	Φ	∞

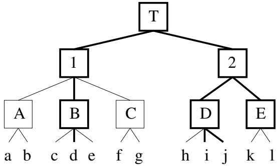
Example



Extract	Insert	Priority queue	Answer	$d_{k=2}$
-	T	$(T, 0)$	Φ	∞
T	1, 2	$(2, 0)$, $(1, 4)$	Φ	∞
2	D, E	$(D, 0)$, $(E, 2)$, $(1, 4)$	Φ	∞
D	h, i, j	$(E, 2)$, $(1, 4)$, $(j, 6)$, $(h, 27)$, $(i, 30)$	Φ	∞
E	k, l	$(1, 4)$, $(j, 6)$, $(h, 27)$, $(i, 30)$, $(k, 36)$, $(l, 38)$	Φ	∞

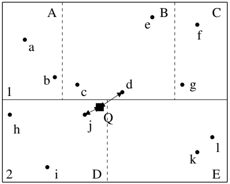
Example

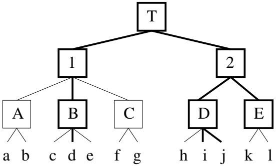




Extract	Insert	Priority queue	Answer	$d_{k=2}$
-	T	$(T, 0)$	Φ	∞
T	1, 2	$(2, 0)$, $(1, 4)$	Φ	∞
2	D, E	$(D, 0)$, $(E, 2)$, $(1, 4)$	Φ	∞
D	h, i, j	$(E, 2)$, $(1, 4)$, $(j, 6)$, $(h, 27)$, $(i, 30)$	Φ	∞
E	k, l	$(1, 4)$, $(j, 6)$, $(h, 27)$, $(i, 30)$, $(k, 36)$, $(l, 38)$	Φ	∞
1	A, B, C	$(B, 4)$, $(j, 6)$, $(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(k, 36)$, $(l, 38)$	Φ	∞

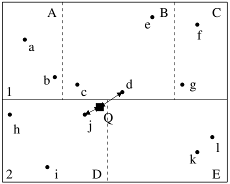
Example

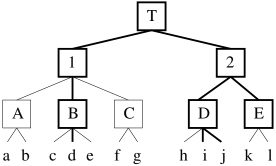




Extract	Insert	Priority queue	Answer	$d_{k=2}$
-	T	$(T, 0)$	Φ	∞
T	1, 2	$(2, 0)$, $(1, 4)$	Φ	∞
2	D, E	$(D, 0)$, $(E, 2)$, $(1, 4)$	Φ	∞
D	h, i, j	$(E, 2)$, $(1, 4)$, $(j, 6)$, $(h, 27)$, $(i, 30)$	Φ	∞
E	k, l	$(1, 4)$, $(j, 6)$, $(h, 27)$, $(i, 30)$, $(k, 36)$, $(l, 38)$	Φ	∞
1	A, B, C	$(B, 4)$, $(j, 6)$, $(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(k, 36)$, $(l, 38)$	Φ	∞
B	c, d, e	$(j, 6)$, $(d, 8)$, $(c, 11)$, $(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(e, 32)$, $(k, 36)$, $(l, 38)$	Φ	∞

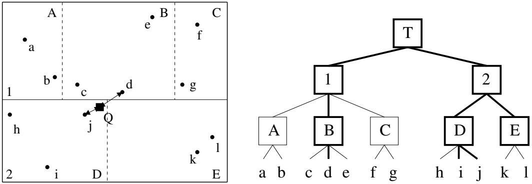
Example (contd.)





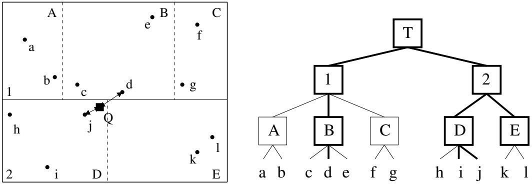
Extract	Insert	Priority queue	Answer	$d_{k=2}$

Example (contd.)



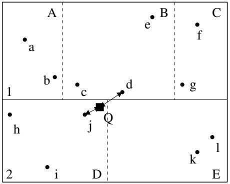
Extract	Insert	Priority queue	Answer	$d_{k=2}$
B	c, d, e	$(j, 6)$, $(d, 8)$, $(c, 11)$, $(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(e, 32)$, $(k, 36)$, $(l, 38)$	Φ	∞

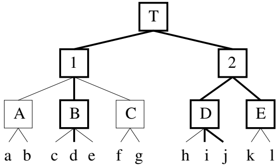
Example (contd.)



Extract	Insert	Priority queue	Answer	$d_{k=2}$
B	c, d, e	$(j,6)$, $(d,8)$, $(c,11)$, $(A,12)$, $(C,22)$, $(h,27)$, $(i,30)$, $(e,32)$, $(k,36)$, $(l,38)$	Φ	∞
j	-	$(d,8)$, $(c,11)$, $(A,12)$, $(C,22)$, $(h,27)$, $(i,30)$, $(e,32)$, $(k,36)$, $(l,38)$	$\{j\}$	∞

Example (contd.)





Extract	Insert	Priority queue	Answer	$d_{k=2}$
B	c, d, e	$(j, 6)$, $(d, 8)$, $(c, 11)$, $(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(e, 32)$, $(k, 36)$, $(l, 38)$	Φ	∞
j	-	$(d, 8)$, $(c, 11)$, $(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(e, 32)$, $(k, 36)$, $(l, 38)$	$\{j\}$	∞
d	-	$(c, 11)$, $(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(e, 32)$, $(k, 36)$, $(l, 38)$	$\{j, d\}$	8

Incremental nearest neighbors

- Obtain 10 nearest neighbors
- Not satisfied with the results
- So, want 5 more nearest neighbors

Incremental nearest neighbors

- Obtain 10 nearest neighbors
- Not satisfied with the results
- So, want 5 more nearest neighbors
- Issue a 10-NN query and then a 15-NN query

Incremental nearest neighbors

- Obtain 10 nearest neighbors
- Not satisfied with the results
- So, want 5 more nearest neighbors
- Issue a 10-NN query and then a 15-NN query
- Not efficient since answer to 15-NN query contains results from 10-NN query

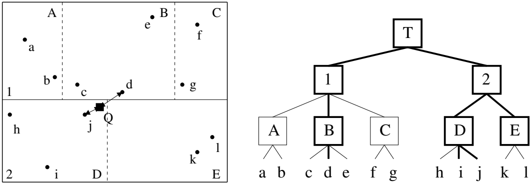
Incremental nearest neighbors

- Obtain 10 nearest neighbors
- Not satisfied with the results
- So, want 5 more nearest neighbors
- Issue a 10-NN query and then a 15-NN query
- Not efficient since answer to 15-NN query contains results from 10-NN query
- Idea is to reuse the information collected earlier

Incremental nearest neighbors

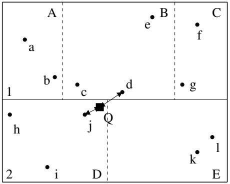
- Obtain 10 nearest neighbors
- Not satisfied with the results
- So, want 5 more nearest neighbors
- Issue a 10-NN query and then a 15-NN query
- Not efficient since answer to 15-NN query contains results from 10-NN query
- Idea is to reuse the information collected earlier
- Incremental nearest neighbors are best solved by *best-first search*

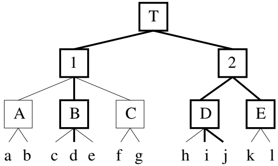
Example



Extract	Insert	Priority queue	Answer	$d_{k=4}$

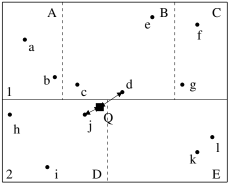
Example

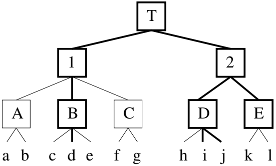




Extract	Insert	Priority queue	Answer	$d_{k=4}$
d	-	$(c, 11)$, $(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(e, 32)$, $(k, 36)$, $(l, 38)$	$\{j, d\}$	∞

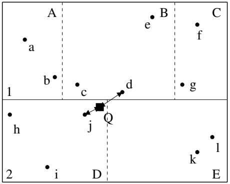
Example

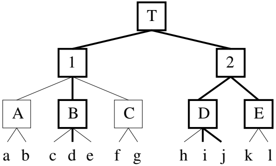




Extract	Insert	Priority queue	Answer	$d_{k=4}$
d	-	$(c, 11)$, $(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(e, 32)$, $(k, 36)$, $(l, 38)$	$\{j, d\}$	∞
c	-	$(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(e, 32)$, $(k, 36)$, $(l, 38)$	$\{j, d, c\}$	∞

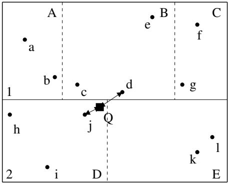
Example

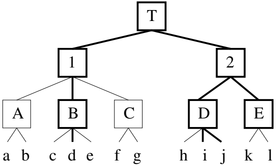




Extract	Insert	Priority queue	Answer	$d_{k=4}$
d	-	$(c, 11)$, $(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(e, 32)$, $(k, 36)$, $(l, 38)$	$\{j, d\}$	∞
c	-	$(A, 12)$, $(C, 22)$, $(h, 27)$, $(i, 30)$, $(e, 32)$, $(k, 36)$, $(l, 38)$	$\{j, d, c\}$	∞
A	a, b	$(b, 15)$, $(C, 22)$, $(a, 26)$, $(h, 27)$, $(i, 30)$, $(e, 32)$, $(k, 36)$, $(l, 38)$	$\{j, d, c\}$	∞

Example

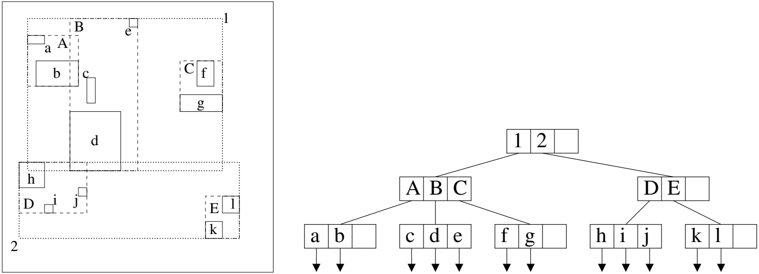




Extract	Insert	Priority queue	Answer	$d_{k=4}$
d	-	$(c,11)$, $(A,12)$, $(C,22)$, $(h,27)$, $(i,30)$, $(e,32)$, $(k,36)$, $(l,38)$	$\{j,d\}$	∞
c	-	$(A,12)$, $(C,22)$, $(h,27)$, $(i,30)$, $(e,32)$, $(k,36)$, $(l,38)$	$\{j,d,c\}$	∞
A	a, b	$(b,15)$, $(C,22)$, $(a,26)$, $(h,27)$, $(i,30)$, $(e,32)$, $(k,36)$, $(l,38)$	$\{j,d,c\}$	∞
b	-	$(C,22)$, $(a,26)$, $(h,27)$, $(i,30)$, $(e,32)$, $(k,36)$, $(l,38)$	$\{j,d,c,b\}$	15

R-tree

- Multi-dimensional B+-tree
- Can store hyper-dimensional points, lines, shapes, etc.
- Objects bounded by a hyper-dimensional minimum bounding rectangle (MBR)
- In d dimensions, hyper-rectangle is specified by $2d$ parameters
- 2 in each dimension to indicate the minimum and maximum values
- Data are contained in leaves only which are all at the same level
- Does *not* index *dead space*
- Children *may* overlap



Structure of an R-tree node

- Design is based on disk page size
- *Underflow* and *overflow* parameters α and β
- α guarantees space utilization (is 2 for root)
- $\alpha \leq \left\lceil \frac{\beta}{2} \right\rceil$:

Structure of an R-tree node

- Design is based on disk page size
- *Underflow* and *overflow* parameters α and β
- α guarantees space utilization (is 2 for root)
- $\alpha \leq \left\lceil \frac{\beta}{2} \right\rceil$: when two underflowing nodes merge, they do not overflow

Structure of an R-tree node

- Design is based on disk page size
- *Underflow* and *overflow* parameters α and β
- α guarantees space utilization (is 2 for root)
- $\alpha \leq \left\lceil \frac{\beta}{2} \right\rceil$: when two underflowing nodes merge, they do not overflow
- Disk page capacity is C bytes
- Key consumes γ bytes
- Pointer to key (or object) requires η bytes

$$C \geq 2.d.\beta.\gamma + \beta.\eta$$
$$\Rightarrow \beta_{\text{R-tree}} = \left\lfloor \frac{C}{2.d.\gamma + \eta} \right\rfloor$$

Insertion

- Data is inserted at leaves
- Since children overlap, there can be multiple choices

Insertion

- Data is inserted at leaves
- Since children overlap, there can be multiple choices
- Choose child requiring *least volume enlargement*
  - If tie, then child with *least volume*, and then *lesser number of children*

Insertion

- Data is inserted at leaves
- Since children overlap, there can be multiple choices
- Choose child requiring *least volume enlargement*
  - If tie, then child with *least volume*, and then *lesser number of children*
- Ensures least dead space indexing
- If no overflow at leaf, insert
- Otherwise, split
- If overflows continue to root, height gets incremented

Deletion

- Find entry by searching
- Delete entry

Deletion

- Find entry by searching
- Delete entry
- If underflow, *delete* leaf node and put entries into temporary set
- Adjust MBRs of index entries of ancestors till root
- Collect all underflowing entries into the temporary set
- Height may decrease if root underflows

Deletion

- Find entry by searching
- Delete entry
- If underflow, *delete* leaf node and put entries into temporary set
- Adjust MBRs of index entries of ancestors till root
- Collect all underflowing entries into the temporary set
- Height may decrease if root underflows
- *Re-insert* orphan entries from temporary set
- Orphan entries are inserted at the same level

Deletion

- Find entry by searching
- Delete entry
- If underflow, *delete* leaf node and put entries into temporary set
- Adjust MBRs of index entries of ancestors till root
- Collect all underflowing entries into the temporary set
- Height may decrease if root underflows
- *Re-insert* orphan entries from temporary set
- Orphan entries are inserted at the same level
- To avoid costly re-inserts, simply mark as “deleted”

Deletion

- Find entry by searching
- Delete entry
- If underflow, *delete* leaf node and put entries into temporary set
- Adjust MBRs of index entries of ancestors till root
- Collect all underflowing entries into the temporary set
- Height may decrease if root underflows
- *Re-insert* orphan entries from temporary set
- Orphan entries are inserted at the same level
- To avoid costly re-inserts, simply mark as "deleted"
- *Update:*

Deletion

- Find entry by searching
- Delete entry
- If underflow, *delete* leaf node and put entries into temporary set
- Adjust MBRs of index entries of ancestors till root
- Collect all underflowing entries into the temporary set
- Height may decrease if root underflows
- *Re-insert* orphan entries from temporary set
- Orphan entries are inserted at the same level
- To avoid costly re-inserts, simply mark as “deleted”
- *Update:* Delete and re-insert

Splitting a node

- How to distribute $\beta + 1$ entries into two nodes having at least α entries each
- *Search efficiency* to distinguish between two splits

Splitting a node

- How to distribute $\beta + 1$ entries into two nodes having at least α entries each
- *Search efficiency* to distinguish between two splits
- Minimize the *probability* that *both* the new nodes are searched

Splitting a node

- How to distribute $\beta + 1$ entries into two nodes having at least α entries each
- *Search efficiency* to distinguish between two splits
- Minimize the *probability* that *both* the new nodes are searched
- Requires knowledge of query distribution
- Assumed to be uniform across the space

Splitting a node

- How to distribute $\beta + 1$ entries into two nodes having at least α entries each
- *Search efficiency* to distinguish between two splits
- Minimize the *probability* that *both* the new nodes are searched
- Requires knowledge of query distribution
- Assumed to be uniform across the space
- Probability is directly correlated to *volume* of MBR
- Therefore, *minimize sum of volumes*

Splitting a node

- How to distribute $\beta + 1$ entries into two nodes having at least α entries each
- *Search efficiency* to distinguish between two splits
- Minimize the *probability* that *both* the new nodes are searched
- Requires knowledge of query distribution
- Assumed to be uniform across the space
- Probability is directly correlated to *volume* of MBR
- Therefore, *minimize sum of volumes*
- Three different strategies:

Splitting a node

- How to distribute $\beta + 1$ entries into two nodes having at least α entries each
- *Search efficiency* to distinguish between two splits
- Minimize the *probability* that *both* the new nodes are searched
- Requires knowledge of query distribution
- Assumed to be uniform across the space
- Probability is directly correlated to *volume* of MBR
- Therefore, *minimize sum of volumes*
- Three different strategies:
  1. Exhaustive
     - Examine all possibilities
     - Exponential number of possibilities
     - Way too impractical
     - Optimal, though

Splitting a node

- How to distribute $\beta + 1$ entries into two nodes having at least α entries each
- *Search efficiency* to distinguish between two splits
- Minimize the *probability* that *both* the new nodes are searched
- Requires knowledge of query distribution
- Assumed to be uniform across the space
- Probability is directly correlated to *volume* of MBR
- Therefore, *minimize sum of volumes*
- Three different strategies:
  1. Exhaustive
     - Examine all possibilities
     - Exponential number of possibilities
     - Way too impractical
     - Optimal, though
  2. Quadratic
  3. Linear

Quadratic split

- Choose a pair of entries (called *seeds*) that is the *most wasteful*
- Waste for two entries a and b is $w = vol(U) - vol(a) - vol(b)$ where U is the *covering* hyper-rectangle of a and b
- Surely, the most wasteful pair should not be in the same group
- Choosing this requires *quadratic* time

Quadratic split

- Choose a pair of entries (called *seeds*) that is the *most wasteful*
- Waste for two entries a and b is $w = vol(U) - vol(a) - vol(b)$ where U is the *covering* hyper-rectangle of a and b
- Surely, the most wasteful pair should not be in the same group
- Choosing this requires *quadratic* time
- For each non-assigned entry c
  - Compute waste of combining c with both a and b
  - Assign c to the one with lesser waste
  - If tie, then *smaller* overall volume and then *lesser* number of entries
- Choice depends on order of examining c

Quadratic split

- Choose a pair of entries (called *seeds*) that is the *most wasteful*
- Waste for two entries a and b is $w = vol(U) - vol(a) - vol(b)$ where U is the *covering* hyper-rectangle of a and b
- Surely, the most wasteful pair should not be in the same group
- Choosing this requires *quadratic* time
- For each non-assigned entry c
  - Compute waste of combining c with both a and b
  - Assign c to the one with lesser waste
  - If tie, then *smaller* overall volume and then *lesser* number of entries
- Choice depends on order of examining c
- When one group has $\beta - \alpha + 1$ entries, put the rest into the other group to avoid underflow

Linear split

- Seeds are chosen differently
- Rest of the procedure is same

Linear split

- Seeds are chosen differently
- Rest of the procedure is same
- For each dimension, choose entries with *highest minimum* and *lowest maximum* values along that dimension
- Normalize the difference by dividing by the total range along the dimension
- Choose the entries with the *largest difference* as the seeds
- Requires *linear* time

Discussion

- Split time increases with page size

Discussion

- Split time increases with page size
  - β increases

Discussion

- Split time increases with page size
  - β increases
- Insertion time decreases with page size

Discussion

- Split time increases with page size
  - β increases
- Insertion time decreases with page size
  - Lesser number of overflows

Discussion

- Split time increases with page size
  - β increases
- Insertion time decreases with page size
  - Lesser number of overflows
- Deletion time increases with α

Discussion

- Split time increases with page size
  - β increases
- Insertion time decreases with page size
  - Lesser number of overflows
- Deletion time increases with α
  - More re-inserts

Discussion

- Split time increases with page size
  - β increases
- Insertion time decreases with page size
  - Lesser number of overflows
- Deletion time increases with α
  - More re-inserts
- Searching is relatively insensitive to splitting algorithm
  - Therefore, linear is justified

Discussion

- Split time increases with page size
  - β increases
- Insertion time decreases with page size
  - Lesser number of overflows
- Deletion time increases with α
  - More re-inserts
- Searching is relatively insensitive to splitting algorithm
  - Therefore, linear is justified
- R-trees are quite suitable for relational databases

R*-tree

- Three engineering optimizations over R-trees
  1. Insertion
  2. Splitting
  3. Forced re-insertion during overflow

Insertion

- If child pointers are internal nodes
  - Choose subtree with least volume increase (same as R-tree)

Insertion

- If child pointers are internal nodes
  - Choose subtree with least volume increase (same as R-tree)
- If child pointers are leaf nodes (i.e., level is one up from leaves)
  - Choose subtree with *least overlap enlargement*
- For child entries $E_1, \ldots, E_k$, overlap of E_i is $\sum_{j=1, j \neq i}^{k} volume(E_i \cap E_j)$
- Overlap is sum of intersected volumes
- Quadratic amount of computations

Splitting

- Considers 3 parameters:
  1. Sum of volumes
  2. Sum of margins, i.e., lengths of sides of hyper-rectangles
  3. Overlap, i.e., common space (in other words, the intersection of the volumes)

Splitting

- Considers 3 parameters:
  1. Sum of volumes
  2. Sum of margins, i.e., lengths of sides of hyper-rectangles
  3. Overlap, i.e., common space (in other words, the intersection of the volumes)
- Choice of split axis
  - For each axis, sort entries by low values and then high values
  - All possible $(\beta - 2\alpha + 2)$ distributions are considered
  - For each distribution, *goodness value* is computed using the three parameters

Splitting

- Considers 3 parameters:
  1. Sum of volumes
  2. Sum of margins, i.e., lengths of sides of hyper-rectangles
  3. Overlap, i.e., common space (in other words, the intersection of the volumes)
- Choice of split axis
  - For each axis, sort entries by low values and then high values
  - All possible $(\beta - 2\alpha + 2)$ distributions are considered
  - For each distribution, *goodness value* is computed using the three parameters
- Split axis is best chosen using sum of margins
- Groups are best partitioned using overlap (and then sum of volumes to break ties)

Forced re-insertion

- When overflow occurs, splitting is not done immediately
- p entries are re-inserted

Forced re-insertion

- When overflow occurs, splitting is not done immediately
- p entries are re-inserted
- Which p entries?
- For each entry, compute distance of its centroid to centroid of node
- Starting from minimum distance is called far reinsert
- Starting from maximum distance is called close reinsert
- In far reinsert, increase in volume is lesser
- In close reinsert, entries liekly to go to other nodes
- Experimentally, close reinsert better with $p = 30\,\%$

Forced re-insertion

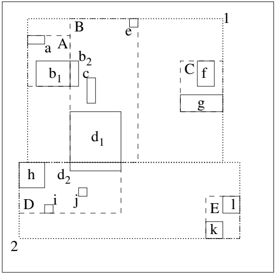
- When overflow occurs, splitting is not done immediately
- p entries are re-inserted
- Which p entries?
- For each entry, compute distance of its centroid to centroid of node
- Starting from minimum distance is called far reinsert
- Starting from maximum distance is called close reinsert
- In far reinsert, increase in volume is lesser
- In close reinsert, entries liekly to go to other nodes
- Experimentally, close reinsert better with $p = 30\,\%$
- Done only once at each level to avoid infinite loop
- In other words, if all entries are inserted in the same node again, splitting is applied

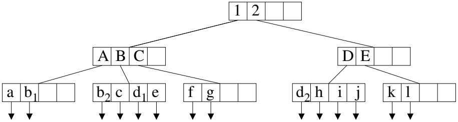
R+-tree

- Mix of R-tree's data partitioning with K-d-B-tree's space-partitioning
- Reduces dead space problem of K-d-B trees by using index rectangles
- Reduces node overlap by making siblings *disjoint*
- Data object may be split into disjoint hyper-rectangles
- Data object may be stored in multiple leaves
- No space utilization guarantee as underflow restriction is relaxed

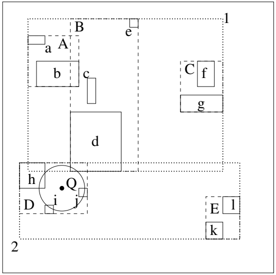
R+-tree

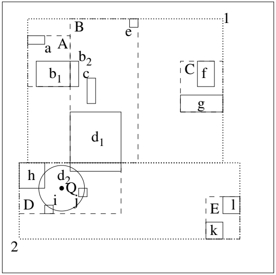
- Mix of R-tree's data partitioning with K-d-B-tree's space-partitioning
- Reduces dead space problem of K-d-B trees by using index rectangles
- Reduces node overlap by making siblings *disjoint*
- Data object may be split into disjoint hyper-rectangles
- Data object may be stored in multiple leaves
- No space utilization guarantee as underflow restriction is relaxed
- Data object b broken into b_1 and b_2
- As a result, nodes A and B do not overlap





Searching

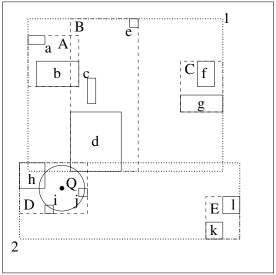




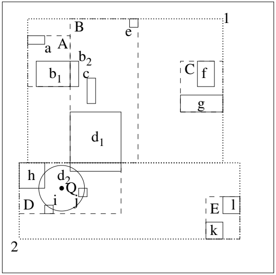
R-tree R+-tree

- Consider query Q with range
- In R-tree, both nodes 1 and 2 are searched
- In R+-tree, only node 2 is searched

Searching



R-tree



R+-tree

- Consider query Q with range
- In R-tree, both nodes 1 and 2 are searched
- In R+-tree, only node 2 is searched
- Trade-off between overlapped searches and increased height
- Not much better than R*-trees
- Maintenance is harder in R+-trees

Hilbert R-tree

- A mix of B+-tree and R-tree
- An ordering on data hyper-rectangles that guides how splits are handled

Hilbert R-tree

- A mix of B+-tree and R-tree
- An ordering on data hyper-rectangles that guides how splits are handled
- A Hilbert space-filling curve grid is imposed on the space
- For a data object, the *Hilbert key* is that of its center
- For an internal node, the *Hilbert key* is the *largest* Hilbert key among its children

Splitting

- Instead of an 1-to-2 split, Hilbert R-tree follows an s-to-$(s+1)$ split
- Each node designates $s-1$ sibling nodes as co-operating sibling nodes

Splitting

- Instead of an 1-to-2 split, Hilbert R-tree follows an s-to-$(s + 1)$ split
- Each node designates $s - 1$ sibling nodes as co-operating sibling nodes
- When a node overflows, its contents are pushed into its co-operating sibling nodes

Splitting

- Instead of an 1-to-2 split, Hilbert R-tree follows an s-to-$(s+1)$ split
- Each node designates $s-1$ sibling nodes as co-operating sibling nodes
- When a node overflows, its contents are pushed into its co-operating sibling nodes
- Only when all such s co-operating sibling nodes overflow, the entries in them are re-assigned to $s+1$ new nodes

Splitting

- Instead of an 1-to-2 split, Hilbert R-tree follows an s-to-$(s+1)$ split
- Each node designates $s-1$ sibling nodes as co-operating sibling nodes
- When a node overflows, its contents are pushed into its co-operating sibling nodes
- Only when all such s co-operating sibling nodes overflow, the entries in them are re-assigned to $s+1$ new nodes
- Increasing s increases space utilization but simultaneously increases the insertion cost
- Overall, $s=2$, i.e., a 2-to-3 split, is best empirically

Discussion

- During insertion, node to insert is chosen as the one having the Hilbert key value just greater than the Hilbert key of the entry

Discussion

- During insertion, node to insert is chosen as the one having the Hilbert key value just greater than the Hilbert key of the entry
- This ensures that the increase in volume of the node where entry gets inserted is low

Discussion

- During insertion, node to insert is chosen as the one having the Hilbert key value just greater than the Hilbert key of the entry
- This ensures that the increase in volume of the node where entry gets inserted is low
- This ordering is similar to that in a B+-tree

Discussion

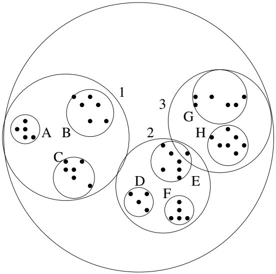
- During insertion, node to insert is chosen as the one having the Hilbert key value just greater than the Hilbert key of the entry
- This ensures that the increase in volume of the node where entry gets inserted is low
- This ordering is similar to that in a B+-tree
- Performs better than R*-tree when data is skewed

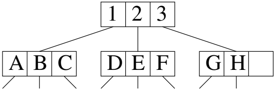
Discussion

- During insertion, node to insert is chosen as the one having the Hilbert key value just greater than the Hilbert key of the entry
- This ensures that the increase in volume of the node where entry gets inserted is low
- This ordering is similar to that in a B+-tree
- Performs better than R*-tree when data is skewed
- Performance deteriorates rapidly for higher dimensions as the property of nearby objects in space enjoying proximity in the Hilbert key ordering gets less respected

SS-tree

- Uses minimum bounding spheres (MBS) instead of MBRs
- Motivated by range and kNN queries which are hyper-spheres
- Center of hyper-sphere is centroid of hyper-spheres representing children nodes
- Radius is the tightest one that covers all children
- Maintains total number of points in the subtree
- Higher fanout due to smaller storage requirements: $d + 1$ parameters instead of $2d$ per child





Bounding spheres and bounding rectangles

- Children often have considerable volume overlap

Bounding spheres and bounding rectangles

- Children often have considerable volume overlap
- Split axis is chosen based on *variance*; otherwise, same as R*-tree

Bounding spheres and bounding rectangles

- Children often have considerable volume overlap
- Split axis is chosen based on *variance*; otherwise, same as R*-tree
- SS-tree has regions with shorter diameters
- R*-tree has regions with smaller volumes

Bounding spheres and bounding rectangles

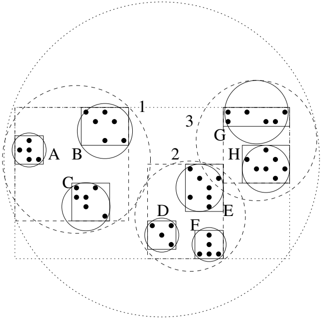
- Children often have considerable volume overlap
- Split axis is chosen based on *variance*; otherwise, same as R*-tree
- SS-tree has regions with shorter diameters
- R*-tree has regions with smaller volumes
- Rectangles in d dimensions
  - Diameter: At most $\sqrt{d}$
  - Volume: At most 1
- Spheres in d dimensions
  - Diameter: At most $\sqrt{d}$
  - Volume: At most $(\sqrt{d}/2)^d(\pi^{d/2}/\Gamma(d/2+1)) \gg 1$

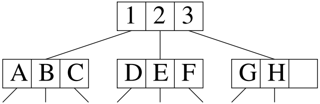
Bounding spheres and bounding rectangles

- Children often have considerable volume overlap
- Split axis is chosen based on *variance*; otherwise, same as R*-tree
- SS-tree has regions with shorter diameters
- R*-tree has regions with smaller volumes
- Rectangles in d dimensions
  - Diameter: At most $\sqrt{d}$
  - Volume: At most 1
- Spheres in d dimensions
  - Diameter: At most $\sqrt{d}$
  - Volume: At most $(\sqrt{d}/2)^d(\pi^{d/2}/\Gamma(d/2+1)) \gg 1$
- Ideally, an index node should combine short diameter regions of SS-tree with small volume regions of R*-tree

SR-tree

- Uses *both* bounding spheres and bounding rectangles
- Index node is *intersection* of these two
  - Tighter
  - Index is implicitly stored
  - Explicitly, both the geometries are stored
- Also maintains number of data objects per child entry





Structure and algorithms

- Center is weighted centroid of child nodes
- Radius is minimum of maximum distances to
  1. Bounding spheres of child nodes
  2. Bounding rectangles of child nodes
- Radius is tighter than SS-tree

Structure and algorithms

- Center is weighted centroid of child nodes
- Radius is minimum of maximum distances to
  1. Bounding spheres of child nodes
  2. Bounding rectangles of child nodes
- Radius is tighter than SS-tree
- Hyper-rectangle is same as R*-tree

Structure and algorithms

- Center is weighted centroid of child nodes
- Radius is minimum of maximum distances to
  1. Bounding spheres of child nodes
  2. Bounding rectangles of child nodes
- Radius is tighter than SS-tree
- Hyper-rectangle is same as R*-tree
- When searching, minimum distance of query point to a node is

Structure and algorithms

- Center is weighted centroid of child nodes
- Radius is minimum of maximum distances to
  1. Bounding spheres of child nodes
  2. Bounding rectangles of child nodes
- Radius is tighter than SS-tree
- Hyper-rectangle is same as R*-tree
- When searching, minimum distance of query point to a node is *maximum* of minimum distances to
  1. Bounding sphere of the node
  2. Bounding rectangle of the node
- Similarly, maximum distance is *minimum* of maximum's
- Produces better lower and upper bounds and, thus, better pruning

Structure and algorithms

- Center is weighted centroid of child nodes
- Radius is minimum of maximum distances to
  1. Bounding spheres of child nodes
  2. Bounding rectangles of child nodes
- Radius is tighter than SS-tree
- Hyper-rectangle is same as R*-tree
- When searching, minimum distance of query point to a node is *maximum* of minimum distances to
  1. Bounding sphere of the node
  2. Bounding rectangle of the node
- Similarly, maximum distance is *minimum* of maximum's
- Produces better lower and upper bounds and, thus, better pruning
- Construction time is larger

Performance

- Storage cost is higher

Performance

- Storage cost is higher
    - Roughly 3 times that of SS-tree, and 1.5 times that of R*-tree

Performance

- Storage cost is higher
  - Roughly 3 times that of SS-tree, and 1.5 times that of R*-tree
- Fanout is lower
- Height is larger
- Number of internal node accesses is more
- Number of leaf node accesses is less
  - Better pruning of index nodes
- Together, faster search than SS-trees and R*-trees

Performance

- Storage cost is higher
  - Roughly 3 times that of SS-tree, and 1.5 times that of R*-tree
- Fanout is lower
- Height is larger
- Number of internal node accesses is more
- Number of leaf node accesses is less
  - Better pruning of index nodes
- Together, faster search than SS-trees and R*-trees
- Mostly, hyper-rectangles are more useful especially at higher dimensions

Fanout example

- What is the maximum fanout of an internal node for an R*-tree, an SS-tree and an SR-tree indexing 5-dimensional values of 8 bytes each with child pointers of size 4 bytes for a page size of 4 KB? Assume that integers require 4 bytes of storage

Tree	R*-tree	SS-tree	SR-tree
Child pointer	4	4	4
Bounding rectangle	$2 \times 5 \times 8 = 80$	-	$2 \times 5 \times 8 = 80$
Bounding sphere	-	$5 \times 8 + 8 = 48$	$5 \times 8 + 8 = 48$
Number of objects	-	4	4
Total size	84	56	136
Fanout	$\lfloor 4096/84 \rfloor = 48$	$\lfloor 4096/56 \rfloor = 73$	$\lfloor 4096/136 \rfloor = 30$

P-tree

- Index is an arbitrary *bounding polyhedron* defined by several hyperplanes
- Hyperplanes are normal to only certain *orientation vectors*
- Number of orientation vectors defines the size of each index entry

P-tree

- Index is an arbitrary *bounding polyhedron* defined by several hyperplanes
- Hyperplanes are normal to only certain *orientation vectors*
- Number of orientation vectors defines the size of each index entry
- Basis vectors along with diagonals is an useful choice
- $2d$ orientation vectors

P-tree

- Index is an arbitrary *bounding polyhedron* defined by several hyperplanes
- Hyperplanes are normal to only certain *orientation vectors*
- Number of orientation vectors defines the size of each index entry
- Basis vectors along with diagonals is an useful choice
- $2d$ orientation vectors
- Map convex polyhedron from d-dimensional *attribute space* to hyper-rectangle at m-dimensional *orientation space*
- Polyhedra intersect if and only if corresponding hyper-rectangles intersect

Bulk-loading

- Inserting data one by one is not good

Bulk-loading

- Inserting data one by one is not good
- Poor space utilization
- Poor organization of objects
- More overlaps
- More dead space

Bulk-loading

- Inserting data one by one is not good
- Poor space utilization
- Poor organization of objects
- More overlaps
- More dead space
- Bulk loading methods consider the entire data at once

Bulk-loading

- Inserting data one by one is not good
- Poor space utilization
- Poor organization of objects
- More overlaps
- More dead space
- Bulk loading methods consider the entire data at once
- Bottom-up methods decide on the layout of leaves first

Bulk-loading

- Inserting data one by one is not good
- Poor space utilization
- Poor organization of objects
- More overlaps
- More dead space
- Bulk loading methods consider the entire data at once
- Bottom-up methods decide on the layout of leaves first
- n objects
- Disk page has capacity for c objects
- Least number of data pages is $p = \lceil n/c \rceil$
- Pack into p leaves
- For next level, pack these p leaves into $\lceil p/c \rceil$ pages, and so on

Packing methods

- Order objects by the first dimension
- Pick c at a time to fill a leaf

Packing methods

- Order objects by the first dimension
- Pick c at a time to fill a leaf
- Ambiguous for non-point objects

Packing methods

- Order objects by the first dimension
- Pick c at a time to fill a leaf
- Ambiguous for non-point objects
- Center, i.e., mean of two extremes

Packing methods

- Order objects by the first dimension
- Pick c at a time to fill a leaf
- Ambiguous for non-point objects
- Center, i.e., mean of two extremes
- Can be ordered by Hilbert key

Packing methods

- Order objects by the first dimension
- Pick c at a time to fill a leaf
- Ambiguous for non-point objects
- Center, i.e., mean of two extremes
- Can be ordered by Hilbert key
- *Clustering* algorithms are good examples as well

STR algorithm

- Sort-tile-recursive (STR)

STR algorithm

- Sort-tile-recursive (STR)
- Consider two-dimensional space
  - Objects are first ordered by x
  - Range of x dimension sliced into $s = \lceil \sqrt{p} \rceil = \lceil \sqrt{n/c} \rceil$ partitions
  - Vertical split axes are chosen in a manner such that each partition (tile) contains $\lceil n/s \rceil$ objects

STR algorithm

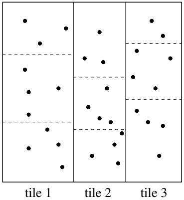
- Sort-tile-recursive (STR)
- Consider two-dimensional space
  - Objects are first ordered by x
  - Range of x dimension sliced into $s = \lceil \sqrt{p} \rceil = \lceil \sqrt{n/c} \rceil$ partitions
  - Vertical split axes are chosen in a manner such that each partition (tile) contains $\lceil n/s \rceil$ objects
  - Each such partition is partitioned according to y dimension
  - s partitions with $\lceil n/s \rceil$ objects per tile

STR algorithm

- Sort-tile-recursive (STR)
- Consider two-dimensional space
  - Objects are first ordered by x
  - Range of x dimension sliced into $s = \lceil \sqrt{p} \rceil = \lceil \sqrt{n/c} \rceil$ partitions
  - Vertical split axes are chosen in a manner such that each partition (tile) contains $\lceil n/s \rceil$ objects
  - Each such partition is partitioned according to y dimension
  - s partitions with $\lceil n/s \rceil$ objects per tile
  - Each page contains at most $\lceil n/s^2 \rceil = \lceil n/p \rceil = c$ objects

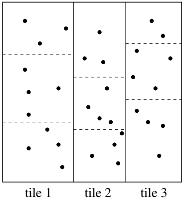
STR algorithm

- Sort-tile-recursive (STR)
- Consider two-dimensional space
  - Objects are first ordered by x
  - Range of x dimension sliced into $s = \lceil \sqrt{p} \rceil = \lceil \sqrt{n/c} \rceil$ partitions
  - Vertical split axes are chosen in a manner such that each partition (tile) contains $\lceil n/s \rceil$ objects
  - Each such partition is partitioned according to y dimension
  - s partitions with $\lceil n/s \rceil$ objects per tile
  - Each page contains at most $\lceil n/s^2 \rceil = \lceil n/p \rceil = c$ objects
- Pages are made *disjoint*



STR algorithm

- Sort-tile-recursive (STR)
- Consider two-dimensional space
  - Objects are first ordered by x
  - Range of x dimension sliced into $s = \lceil \sqrt{p} \rceil = \lceil \sqrt{n/c} \rceil$ partitions
  - Vertical split axes are chosen in a manner such that each partition (tile) contains $\lceil n/s \rceil$ objects
  - Each such partition is partitioned according to y dimension
  - s partitions with $\lceil n/s \rceil$ objects per tile
  - Each page contains at most $\lceil n/s^2 \rceil = \lceil n/p \rceil = c$ objects
- Pages are made *disjoint*



- In d dimensions, $s' = \lceil p^{\frac{1}{d}} \rceil$ partitions are created
- Each slice contains $n/s' = c.p^{\frac{d-1}{d}}$ objects in $(d-1)$ dimensions