Indexing(B-tree,R-tree)

Lee,Jae-Yong
October 10,2002

Presentation Outline

- Motivation
- B tree
- B+ tree, B* tree
- R tree
- R* tree
- SS tree,SR tree

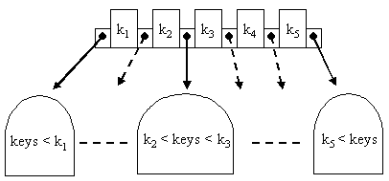
Why indexing?

- When data is too large to fit in memory, then the number of disk accesses is important.
- A disk access is expensive and worth about 200,000 instructions.
- To minimize disk accesses

m-way search tree

- A generalization of binary search tree
- Each node has at most m child-pointers
- If k<=m is the number of children, then the node has exactly k-1 keys.
- The tree is ordered in keys.

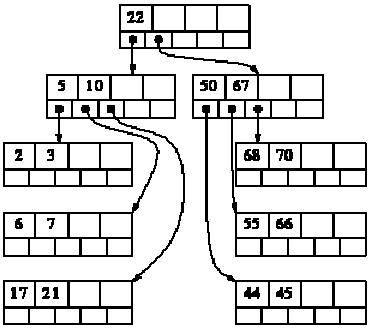
Multiway Search Trees (cont.)



B tree(Definition)

- B-tree of order 2d is an (2d+1)-way search tree such that:
  - all leaf nodes are on the same level.
  - completely balanced.
  - All non-leaf nodes(except root node) have at most 2d keys and 2d +1 pointers and at least d keys and d+1 pointers.
  - The root has at least one key and 2 child pointers.

Example(B-tree)



Height of B-tree

- What is the maximum height of a B-tree with n keys?
- The maximum height of a B-tree will give an upper bound on the number of disk access.
- The minimum number of keys in a B-tree of order 2d and depth h :

$$1 + 2d + 2d(d+1) + 2d(d+1)^2 + \sim + 2d(d+1)^{h-1}$$

$$= 1 + \sum_{i=1}^{h} 2d(d+1)^{i-1}$$

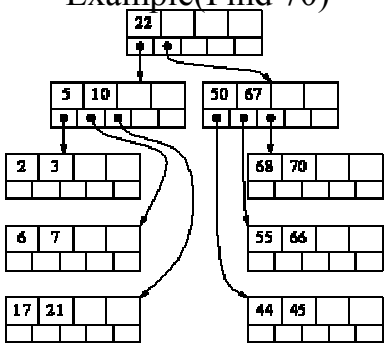
- The maximum height of a B-tree with n keys:

$$\log_{d+1} \frac{n}{2d} = O(\log_d n)$$

Search(Find)

- B-tree is always completely balanced.
- The length of retrieval path is at most the height of the B-tree.
- The cost of a search is $O(\log_d n)$.

Example(Find 70)



Insert

- First, a find proceeds from the root to locate the proper leaf node for insertion.
  => $O(\log_d n)$
- If the leaf node is already full, split it.
  Otherwise, done.

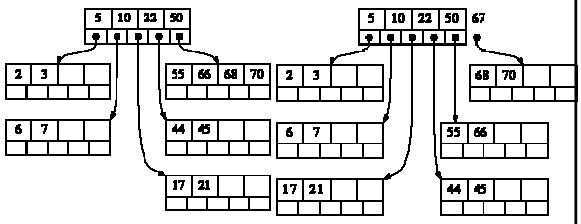
Splitting a node

- 2d+1 keys are split to 2 new nodes.
- One node has the smallest d keys and the other has the largest d keys.
- A separator key is promoted to the parent node.
- If the parent node is already full, the parent node is split too.

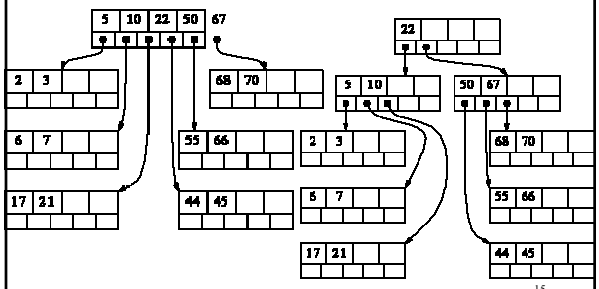
Splitting a node(cont.)

- In the worst case, splitting propagates to the root node and the tree increases in the height by one level.

Example(Splitting,Insert 67)



Example(cont.)



Insert cost

- Find operation : $O(\log_d n)$
- Splitting : $O(\log_d n)$
- Total cost is at most doubled as a find .
- Usually, nodes in the path reside in memory after the find operation.

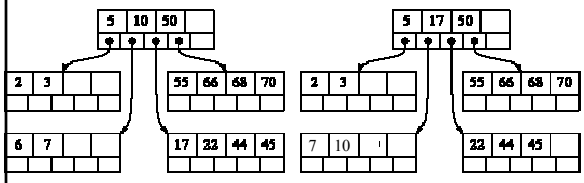
Delete

- First, a find operation to locate the node
- The key can be found in either a leaf or a nonleaf
- In Insert, the key is always inserted to a leaf node
- Nonleaf case
  - Find an adjacent key in a subtree of the deleted key (found in a left-most leaf node of the right subtree)
  - Move it to the deleted position
  - Process in the leaf node

Underflow during delete

- Underflow
  - Less than d keys occupy the node(d-1 keys)
  - Need to redistribute the keys to a neighbor node(including their parent key)
- Redistribution
  - Evenly divide keys between two neighbors
  - Need at least 2d+1 (d+d+1) keys to redistribute
  - What if the neighbor node does not have enough keys (has just d nodes) -> concatenation

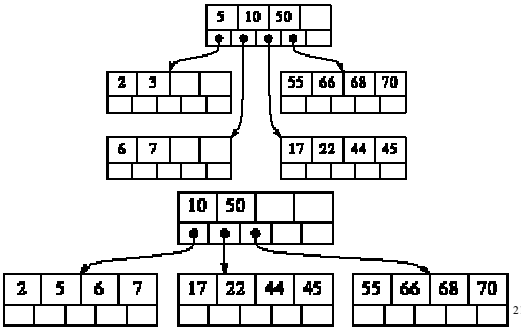
Redistribution(Delete 6)



Underflow during delete(cont.)

- Concatenation
  - Inverse of Splitting
  - Keys in two neighbor nodes and a separator key in the parent node
  - The merged node has 2d keys(full).
  - The parent node may be concatenated.
    - May be propagated all the way to the root node
    - Decreasing the height of the B-tree

Concatenation(delete 3)



Delete cost

- Same as Insert operation

Running time

- Running time of each operation is directly related to the height of the tree
- The height of a B-tree depends on :
  - the number of keys in the tree
  - the branching factor(2d+1)
    - Practical limit on a size of node
    - Usually a block size(page size)
  - the storage utilization
    - B* tree(at least 2/3 of a node is full)

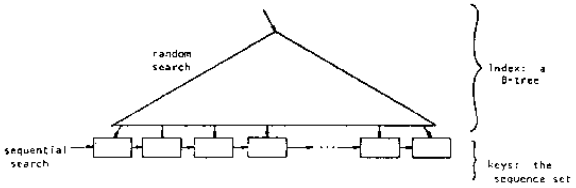
Sequential processing in a B-tree

- Preorder tree walk
- Stacks the nodes along a path from the root
- Require space for at least the height
- B-tree does not do well in a sequential processing
- B+ tree

B+ tree

- Independent index and sequence sets
- Sequence set(leaves)
  - all actual keys reside in the leaves.
  - leaf nodes are linked for a sequential processing
- Index set(non-leaves)
  - Just a roadmap to reach a leaf node
  - Not necessarily actual keys
- Better sequential processing

Example(B+ tree)



B* tree

- Each node is at least 2/3 full.
- Increasing storage utilization
  - speeding up search operations
- Split
  - Redistribution
    - Delay the split until two siblings are full
    - Reduce the number of splits
  - 2 nodes are divided into 3 nodes

R-Tree(Motivation)

- Classical indexing (like B-Trees) are not suitable for multidimensional spatial searching

Spatial Objects

- Data objects of non-zero size in multi-dimensional spaces
  - ex) a country in a map
- Queries are also spatial
  - ex) Find all objects in an area

Nodes in R-tree

- Leaf node
  - has entries of the form (I, tuple-identifier)
    - tuple-identifier: a reference to an object in DB
    - $I = (I_0, I_1, ..., I_{n-1})$: an n-dimensional bounding box
- Nonleaf node
  - has entries of the form (I,child-pointer)
    - child-pointer
    - I : covers all rectangles in the lower node's entries

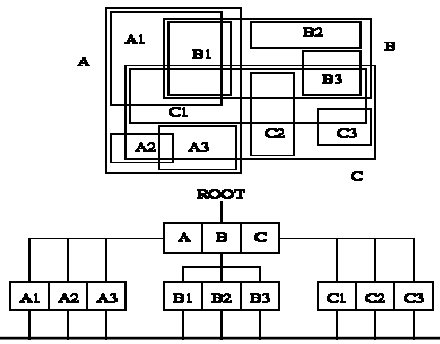
R-tree

- M is the max number of entries in a node.
- $m \le \frac{M}{2}$ is the min number of entries
- m is not fixed and varies for performance tuning.

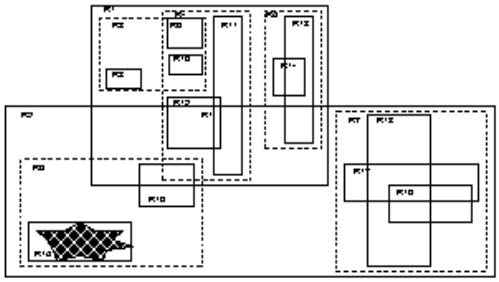
R-tree(Definition)

- Every node contains between m and M entries.(except the root)
- The root has at least two children.
- All leaves appear on the same level
  - completely balanced
- I is a rectangle that contains all objects in lower levels

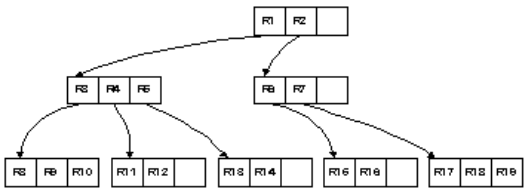
Example



An Example



An Example



R-tree

- Nonleaf nodes serve as a set of indices.
  - B+ tree
- Bounding boxes in a same level are not disjoint.
  - A level in B-tree consists of disjoint intervals.
- Each entry in a node represents an area in space.

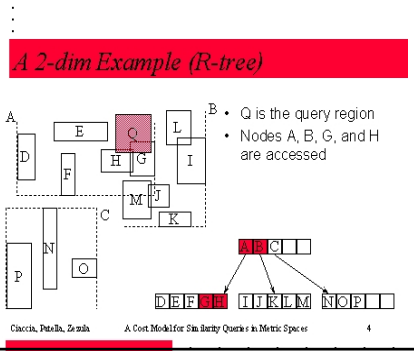
Search

- Bounding boxes in each level may be overlapped.
  - More than one subtrees under a node may need to be searched
  - No good worst case performance
  - Need some techniques to minimize searches
    - Reduce dead-space
  - Update algorithms maintains “good” trees.

Search(cont.)

- Query : Search rectangle S
- Search all children if their bounding boxes overlap with S.
- All leaf nodes have a tuple identifier
- The cost depends on the structure of the tree
  - Need to minimize the area of MBR of nodes

Example(Search)



Insert

- Like the B-tree, sometimes it is necessary to split a node
- Since the decision whether to visit a node depends on whether its covering rectangle overlaps the search area, areas of two new nodes should be minimized
  - All approaches of optimizing the retrieval performance have to be applied during Insert operation

Insert(cont.)

- Similar to B-tree in handling split and simple insertions
- ChooseLeaf: selects a leaf node L in which to place the new entry
- The algorithm tries to find the region whose rectangle needs least enlargement to include the new entry

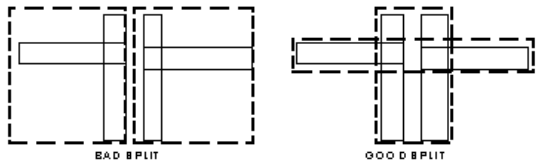
Delete

- Like in B-Tree
  - But find a leaf to be deleted(like B+ tree)
- Sometimes you need to condense a tree due to the elimination of one node

Delete(Condense)

- Underflow in the node after a deletion
- An underflow may be propagated.
- If underflow in the node,N,
  - Delete the index entry of parent node,P
  - Add N to the set of eliminated nodes,Q
  - When the propagation is done, reinsert all entries of nodes in Q
    - Need to reinsert a proper level in the tree

Node Splitting



Node splitting algorithm

- "Good" split can reduce the number of nodes during subsequent searches
  - Exhaustive algorithm
  - Quadratic algorithm(Heuristic)
  - Linear algorithm(Heuristic)

Exhaustive method

- Generate all possible groupings (2^M possibilities)
- M is typically 50
- Too expensive

Quadratic method

- picks 2 of the M+1 entries to be seeds of the 2 new groups -> $O(dM^2)$
  - the pair that would waste the most are if both were put in the same group
- then assign to groups one entry at a time
- at each step the area expansion required to add each remaining entry to group is calculated, and the entry assigned is the one showing the greatest difference between the 2 groups-> $O(dM^2)$

Linear method

- Different seed picking-> $O(dM)$
  - find the entry whose rectangle has the highest low side, and the one with the lowest high side. Record the separations for each dimension ->
  - normalize the separations by dividing by the width of the entire set
  - choose the pair with the greatest separation along any dimension.
- Assign each entry to a group -> $O(dM)$
  - Pick any entry and then assign the group which requires less expansion

Performance Test

- evaluate m (minimum number of entries) as a function of M (maximum number) and the heuristics
- The linear node-split algorithm proved to be as good as more expensive techniques
  - faster and the slightly worse quality of the splits did not affect search performance noticeably

Problems in R-tree

- Just try to minimize the area of each enclosing rectangle in the index nodes.
- What can we consider to produce "good" trees?
  - R* tree
- What if the query is not a rectangle?
  - SS tree, SR tree

R* tree

- Optimization Criteria:
  - (O1) Area covered by an index MBR
  - (O2) Overlap between directory MBR
  - (O3) Margin of a directory rectangle
    - Next slide.
  - (O4) Storage utilization
- Sometimes it is impossible to optimize all the above criteria at the same time

Overlap &Margin

- (O2) Overlap between directory MBR
  - Sum of overlapped areas of all pairs of entries in the MBR
- (O3) Margin of a directory rectangle
  - Margin : sum of lengths of the edges in MBR
  - Minimizing the margin of Rectangle
    - Rectangle can be more quadratic. Quadratic object can be packed easier. ->smaller MBR

R*-tree(cont.)

- ChooseSubtree
  - Leaf node:
    - Least overlap enlargement
    - Least area enlargement(R-tree)
    - Smaller area
  - Nonleaf node
    - Least area enlargement(R-tree)
    - Smaller area

Split

- SplitNode
  - Choose the axis to split
  - Choose 2 groups along the chosen axis
- ChooseSplitAxis
  - Along each axis, sort rectangles and break them into 2 groups(M-2m+2 ways)
  - Compute the sum S of all margin-values of each pair of groups. Choose the one that minimizes S
- ChooseSplitIndex
  - Along the chosen axis, choose the grouping that gives the minimum overlap-value

Forced Reinsert

- The structure of tree is non-deterministic
- Dynamic reorganization.
- Defer splits, by forced-reinsert
  - Instead splitting, temporarily delete some entries, shrink overflowing MBR, and reinsert those entries
- Which ones to re-insert?(next slide)
- How many? 30% (by experiments)

Forced Reinsert(Details)

- Calculate centers of MBRs of all entries in the overflown node and the center of MBR of the node
- Remove p farthest entries from the node
- Then reinsert the entries
- If at least one entry is inserted into different node, no splitting the overflown node.
  - But may cause other nodes to be overflown

SS & SR-tree

- An index structure for high-dimensional nearest neighbor queries
- the similarity queries on feature vectors which is widely used for performing content-based retrieval of images