Datenbanksysteme II:
Multidimensional Index Structures 2

Ulf Leser
Content of this Lecture

• Introduction
• Partitioned Hashing
• Grid Files
• kdb Trees
  – kd Tree
  – kdb Tree
• R Trees
kd Tree

- Grid file disadvantages
  - All hyperregions of the d-dimensional space are eventually split at the same scales (dimension/position)
  - First cell that overflows determines split
  - This choice is global and never undone

- kd Trees
  - Multidimensional variation of binary search trees
  - Hierarchical splitting of space into regions
  - Regions in different subtrees may use different split positions
  - Better adaptation to local clustering of data
  - Note: kd Tree originally is a main memory data structure
General Idea

- **Binary**, rooted tree
- Inner nodes define splits (*dimension / value*)
- Dimensions may be mixed in same level
- Leaves: Values + TIDs
- Each leaf represents *d*-dimensional convex hypercube with *m* border planes (*m ≤ 2d*)
- No balancing
  - Bad WC search
Blocks and Points

- **Keep everything in memory**
  - Leaves are singular points
  - Does not exploit caching / seq. reads

- **Tree in mem and blocks on disk**
  - Splits are delayed until block overflows

- **Store everything on disk**
  - k-DB Tree: Later

- **On modern hardware**
  - Random mem access in inner tree
  - Larger leaves create smaller trees
  - Parallel search? SIMD? Tree layout?
  - BB-Tree: Later
• Every split can be chosen freely within borders defined by parents
• Splits are local
Local Adaptation
Search Operations

- Exact point search
  - ?
- Partial match query
  - ?
- Range query
  - ?
- Nearest Neighborhood
  - ?
Search Operations

• Exact point search (result size 1)
  – In each inner node, decide upon direction based on split condition
  – Search inside leaf
  – Complexity = height of tree = $O(n)$ in worst case

• Partial match query
  – If dimension of condition in inner node is part of the query – proceed as for exact match
  – Otherwise, follow all children (multiple search paths)
  – Worst case (no conditions) searchers entire tree

• Range query
  – Follow all children matching the range conditions (multiple paths)
Nearest Neighbor

- Search point
- Upon descending, build a priority queue of all directions not taken
  - Compute minimal distance between point and hyper-region not followed
  - Keep sorted by this minimal distance
- Once at a leaf, visit hyperregions in order of distance to query point
  - Jump to split point and follow closest path
  - Regions not visited are put into priority queue
  - Iterate until point found such that provably no closer point exists
Example

\[ x < 3 \]
\[ y \geq 1 \]
\[ (2,0) \]

\[ x \geq 3 \]
\[ y < 7 \]
\[ (0,4) \]
\[ (1,1) \]

\[ y \geq 3 \]
\[ x < 5 \]
\[ (3,1) \]

\[ y \geq 2 \]
\[ y < 2 \]
\[ (4,6) \]
\[ (3,3) \]

\[ (6,4) \]

Query: (5.1, 2.2)
kd-Tree Insertion

• Search leaf block; if space available – done
  – The original kd-Tree has no blocks – we always split
• Otherwise, chose split (dimension + position) for this block
  – This is a local decision, valid for subtree of this node
  – Option 1: Use each dimension in turn and split region into two equally sized subspaces (expects uniform distribution)
  – Option 2: Consider current points in leaf and split in two sets of approximately equal size (expects temporally constant distribution)
    • But which dimension?
    • Considering all is expensive – use heuristics
  – Usual problem: We don’t know the future points
  – Wrong decisions in early splits may lead to tree degradation
    • As for Grid-Files, there is no guarantee on fill degree
Deletion

- Search leaf block and delete point
- If block becomes (almost) empty
  - If empty: Remove; else: Do nothing – bad fill degree
  - Merge with neighbor leaf (if existing)
    - Two leaves and one parent node are replaced by one leaf
    - Not very clever if neighbor almost full
  - Balance with neighbor leaf (if existing)
    - Change split condition in parent such that children have equal size
    - Not very clever if neighbor almost empty
  - Consider larger neighborhood: Grant parents, grant-grant-par ...

- kd trees have no guaranteed balance (~ depth)
- There is no guaranteed fill degree
Static kd Trees

- Assume the set of points to be indexed is static and known
- We can build worst-case optimal kd Trees
  - Rotate through dimensions
    - Typically in order of variance – wide spread dimensions first
  - Sort remaining points and choose median as split point
  - Guarantees tree depth of $O(\log(n))$ for point queries
  - But clustering of points not considered – bad similarity queries
    - Nearby points are not nearby in the tree
- Variant (for sim-search): K-means trees
  - Iterative $k$-means clustering of points
  - K: Tree width (fanout)
  - Faster similarity queries, tree depth not guaranteed
Content of this Lecture

• Introduction
• Partitioned Hashing
• Grid Files
• kdb Trees
  – kd Tree
  – kdb Tree
• R Trees
kd Trees on Secondary Storage – Naive Solution

- Store each inner node in one block
  - Inner blocks are essentially empty
  - Since tree is not balanced, worst case requires $O(n)$ IO
Better: Fill Inner Blocks

- Option 1: Build **k-ary kd-Trees**
  - Let inner nodes split **one dimension at many values**
  - When leaf overflows, insert new split into parent
  - When leaf underflows, merge and remove split from parent
  - Still **not balanced**, no guaranteed fill degree

- With skewed data

<table>
<thead>
<tr>
<th>Age</th>
<th>10</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weight</td>
<td>8</td>
<td>15</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Correlated dimensions
**kdb trees**

- **Option 2:** Map many inner nodes to a single blocks
  - Inner nodes have two children (mostly in the same block)
  - Each block holds many inner nodes
  - Inner blocks have many children
    - Roots of kd trees in other blocks
  - Can be balanced (later)
  - No guaranteed fill degree

- **Operations**
  - Searching: As with kd trees, but has guaranteed tree depth
  - Insertion/Deletion: Keep balance
Another View

- Inner blocks define bounding boxes on subtrees
Another View

- Inner blocks define bounding boxes on subtrees
Example – Composite Index

- \(d=3, \ n=1E9, \) block size 4096, \(|\text{point}|=9, \ |\text{b-ptr}|=10\)
  - We need \(~2.2M\) leaf blocks

- **Composite B+ index**
  - Inner blocks store 108-215 pointers; assume optimal density
  - We need 3 levels
    - 2\(^{nd}\) level has 215 blocks and 46,000 pointers
    - 3\(^{rd}\) level has 46K blocks and 10M pointers, 2.2M are needed
  - With uniform distribution, 1st level will mostly split on 1st
dimension, 2nd level on 2nd dimension ...

- **Box query, 5% selectivity in each dimension**
  - We read 5% of 2nd level blocks = 10 IO
  - For each, we read 5% of 3rd level blocks = 107 IO
  - For each, we read 5% of data blocks = 1150 IO
  - Altogether: \(~1250\) IO
Visualization
Example: Partial Box Query

- Box query on 2nd and 3rd dimensions only, asking for a 5% range in both dimensions
  - We need to scan all \(215\) 2nd level blocks
    - Each 2nd level block contains the 5% range of 1st dimension
  - For each, we read 5% of 3rd level blocks = \(2300\) blocks
  - For each, we read 5% of data blocks = \(\sim 25K\) data blocks
  - Altogether: \(26,000\) IO

- Note: 0.05 selectivity in two dimensions means 0.0025 selectivity altogether = 125K points
  - Only \(270\) blocks if optimally packed
With Balanced kdb Tree

- **Balanced kdb tree** will have ~22 levels
  - ~455 points in one block (assume optimal packaging)
  - We need to address $1E9/455 \sim 2^{21}$ blocks
- **Consider 128=2^7 inner nodes in one kdb-block**
  - Rough estimate; we need to store 1 dim indicator, 1 split value, and 2 ptr for each inner node, but most ptr are just offsets into the same block
- **kdb tree structure**
  - 1st level block holds 128 inner nodes = levels 1-7 of kd tree
  - There are 128 2nd level blocks holding levels 8-14 of kd tree
  - There are ~16000 3rd level blocks, each addressing 128 data blocks
Space Covered

- 1st block splits space in 128 regions
- 2nd level block split space in ~16K regions, each region covering 0.00625% of the entire space
- Query selectivity is $(0.05)^3 = 0.000125\%$ of points and of space (given uniform distribution)
- Thus, we very likely find all results in 1 region of the 1st level and in 1 region of the second level
  - In the worst case, we overlap in all dimensions – 8 regions
Box Query Continued

- Box query in all three coordinates, 5% selectivity in each dimension
  - We need to load the root block
  - Very likely, we need to look at only one 2\textsuperscript{nd} level block
  - Very likely, we need to look at only one 3\textsuperscript{rd} level block
  - Assume we need to load all therein addressed 128 data blocks
  - Altogether: $1+1+1+128 = 131$ IO
    - That’s almost optimal
      - But we made many favorable assumptions
      - \textit{kdb-Tree may reach} almost optimal performance
  - Composite index had: $\sim 1250$ IO
Example - Partial Box Query with kdb Tree

- Box query on 2nd and 3rd dimensions only, asking for a 5% range in each dimension
  - In first block (7 levels), we have \( \sim 2 \) splits in each dimension
    - Two times 2 splits, one time three splits
    - Assume we miss the dimension with 3 splits
  - Hence, in \( \sim 4 \) of 7 splits we know where we need to go, in \( \sim 3 \) splits we need to follow both children
  - We need to check only \( 2^3 = 8 \) second-level blocks
    - Again – number gets higher when query range crosses split points
  - Same argument holds in 2nd level blocks = 8*8 data blocks
  - Same argument holds in 3nd level blocks = 8*8*8 data blocks
  - Altogether: 1+8+64+512 \( \sim 580 \) IO
    - Compare to 26,000 for composite index
    - But optimal would be only 270
Balancing upon Insertions

- Similar method as for B+ trees
  - Search appropriate leaf
  - If leaf overflows, split
    - Chose dimension and split value; re-distribute points into two blocks
    - Propagate to parent node
  - In parent node, a leaf must be replaced by an inner node
    - With two new blocks as children
  - This may make the parent overflow – propagate up the tree

- Splitting an inner node
  - Chose a dimension and split value
  - Distribute nodes to two new blocks
    - Split might have to be propagated downwards
    - “Default” split may lead to very bad fill degree
  - Propagate new pointers to parent (and their children)
  - Might lead to reorganization of entire tree
Conclusion

• kdb trees pro
  – Conceptually nice
  – May achieve optimal search performance

• Kdb contra
  – No guaranteed fill degree
    • Many insertions/deletions lead to almost empty leaves
  – Keeping balance requires sporadic tree reorganizations
    • Runtime of single operations become unpredictable

• Nice idea, difficult to implement, rarely used in practice
Content of this Lecture

- Introduction
- Partitioned Hashing
- Grid Files
- kdb Trees
- R Trees
- Conclusions
R-Trees

- Can store geometric objects (with area) as well as points
  - Arbitrary geometric objects are represented by their minimal bounding box (MBB)
- Each object is stored in exactly one region on each level
- Since objects may overlap, regions may overlap
- Only regions containing data objects are represented
  - Allows for fast stop when searching in empty regions
- Tree is kept balanced (like B tree)
- Guaranteed fill degree (like B tree)
- Many variations (see literature)
General Idea

• We group clusters of spatial objects into minimal bounding box (MBB)

• Each MBB is represented by just two corner points
General Idea

- Objects are stored only once in leaf nodes
- We group MBBs hierarchically into overlapping regions
Motivation: Objects that are not points

- We need overlapping regions
  - For instance, if all MBBs overlap
  - No split possible which creates disjoints sets of objects

- Objects crossing a split
  - Stored in only one MBB (R-Tree)
    - Search must examine both
    - No redundant data
  - Stored in both MBB (R+-Tree)
    - Search may choose any one
    - Redundant data
R Tree versus kd Tree
Concepts

- Inner nodes consist of a set of **d-dimensional regions**
  - Every region is a (convex) hypercube - MBB
- Regions are hierarchically organized
- Each region of an inner node points to a subtree or a leaf
- The **region border** is the MBB of all objects in this subtree
  - Inner node: MBB of all child regions
  - Leaf blocks: All objects are contained in the respective region
- Regions in one level may **overlap**
- Regions of a level do not cover the space of its parent completely (as opposed to the KD-tree)
Concepts

• **Guaranteed fill degree**: The number of regions of a node (except for the root) is between m and M
  - M: the maximum number of entries in a node
  - M = \lfloor \text{size}(P) / \text{size}(E) \rfloor \quad \text{P: disk page, E: entry}
  - m: set to some fraction of M

• The root node has at least 2 entries

• **Balanced**: Leaf nodes are at the same level
Searching

- All objects are contained within MBBs
- Thus, a query that does not intersect an MBB cannot intersect the contained objects
- Point query
  - At each inner node, find all regions containing the point
  - Multi-path: All those subtrees must be searched
- Range query: Find all objects (MBBs) overlapping with a given query range (MBB)
  - In each node, intersect query with all regions
  - More than one region might have non-empty overlap
  - All those subtrees must be searched
Example: Searching

No overlap in child regions (only in MBB) – stop search
Inserting an Object

• Traverse the R-tree top-down, starting from the root

• In each node, find all candidate regions
  – Any region may overlap the object completely, partly, or not
  – Object may overlap none, one, or many regions – partly or completely
  – At least one region with complete overlap
    • Choose one (smallest?) and descend
  – None with complete, but at least one with partial overlap
    • Choose one (largest overlap?) and descend
  – No overlapping region at all
    • Choose one (closest?) and descend

• Eventually, we reach a leaf
  – We insert object in only one leaf
Continuation

- If free space in leaf
  - Insert object and adapt MBB of leaf
  - Recursively adapt MBBs up the tree
  - This usually generates larger overlaps – search degrades

- If no free space in leaf
  - Split block in two regions
  - Compute MBBs
  - Adapt parent node: One more child, changed MBBs
  - May affect MBB of higher regions and/or incur overflows at high regions – ascend recursively
Example (from Donald Kossmann)

Compute MBBs for all non-rectangular objects
Example: Insertion, Search Phase

- Search regions whose MBB must be expanded the least
- Repeat on each level
- Here: **Leaf overflow**, split
  - Note: Choosing b4 would avoid split – but how can we know?
Example: Insertion, Split Phase

Several splits are possible
Example: Insertion, Adaptation Phase

- MBBs of all parent nodes must be adapted
- Block split might induce node splits in higher levels of the tree (not here)
Where to Split

- Finding the best splitting strategy has seen ample research
- Option 1: Avoid overlaps
  - Compute split such that overlap is minimal (or even avoided)
  - Minimizes necessity to descend to different children during search
  - May create larger regions – more futile searches in “empty” regions
- Option 2: Minimize space coverage
  - Compute split such that total volume of all MBBs is minimal
  - Increases changes to descend on multiple paths during search
  - But: Unsuccessful searches can stop earlier
Split Strategies

- **Rationale:**
  - Pick two objects as seeds
  - Assign other objects to the closest seeds
Split Strategies

- **Rationale:**
  - Pick two objects as seeds
  - Assign other objects to the closest seeds
  - Closest: the total MBB volume minimally increases
Split Strategies

- **Complexity**
  - Consider a block with \( n \) objects
  - There are \( 2^{n/2} = 2^{n-2} \) possibilities to partition this block into two
  - In multi-dimensional spaces, there is no simple sorting
  - **Use heuristics** instead of optimal solution

- **Original Strategies (Minimizing Overlap)**
  - **Linear**: Pick two pairs with greatest normalized separation. Greedily associate each other object to the region whose space is increased the least
  - **Quadratic**: Pick two pairs such that the two regions minimally overlap and are maximally large. Greedily associate each other object to the region whose space is increased the least
  - **Exponential**: Check all bipartitions and chose the one with minimal overlap
Linear Split

- In each dimension, find two objects with greatest separation
- Normalize the separation by the total extent in that dimension
- Put the two entries E1 and E2 with the greatest normalized separation into different groups
- Greedily associate each other of the M-1 objects to the region whose space is increased the least
Quadratic Split

• Pick the two seed entries E1 and E2 that would waste most area, if put together, that is to maximize:

$$\text{area}(mbb(E1,E2) - \text{area}(E1) - \text{area}(e2))$$

• Complexity: $O((M + 1)^2)$

• Greedily associate each other of the M-1 objects to the region whose space is increased the least
Deletions in the R Tree

• As usual: In case of underflow (<m\% fill degree), the block is removed

• R Trees typically do not move objects to neighbor leaves
  – MBBs would have to be adopted
  – But relationship of MBBs may be quite arbitrary
  – May create very large overlaps, very large spaces covered
  – One could find optimal moves, but ...

• Trick: Delete by Reinsertion
  – Re-Insert every objects that remained in the underflown block
  – Guarantees of the insert strategies will hold
  – No particular delete strategy required – focus on good insertions
  – But costly: A single delete may incur hundreds of inserts
R+ Tree

• Two effects leading to inefficiency during search
  – Overlapping MBBs lead to multiple search paths
  – A few large objects enforce large MBBs covering much dead space

• R+ Tree
  – Objects overlapping with two regions are stored in both (clipping)
  – MBBs in a node never overlap

• Much faster search, but
  – Search must perform duplicate removal as last steps
  – Insertion / deletion may have to walk multiple paths, incurring multiple adaptations
  – Worse space consumption due to redundancy,
  – Insertion may require down- and upward adaption
    • Like kdb Trees
Content of this Lecture

- Introduction
- Partitioned Hashing
- Grid Files
- kdb Trees
- R Trees
- Conclusions
Multidimensional Data Structures Wrap-Up

• Many more MDIS: X tree, VA-file, hb-tree, UB tree, ...
  – Store objects more than once; other than rectangular shapes; map coordinates into integers; ...

• All MDIS degrade with increasing number of dimensions (d>10) or very unusual skew
  – For neighborhood and range queries
  – Hierarchical MDIS degenerate to an expensive linear scan

• Trick: Find lower-dimensional representations with provable lower bounds on distance to prune space
  – Requires distance function-specific lower bounding techniques

• Alternative: Approximate MDIS (LSH, randomized kd Trees)
  – Find almost all neighbors, with/out given probability
Curse of Dimensionality – Consider a growing d

- Consider a typical **rectangular partitioning** method
- Some obvious problems
  - Points need more coordinates, less node capacity – **fan-out decreases**
  - Decreasing fan out – **deeper trees**
  - Just **comparing two points** becomes linearly more expensive
  - Intersecting two objects becomes more expensive
  - These operations are performed all the time when searching and inserting / deleting objects
Curse of Dimensionality – Consider a growing d

- Some less obvious mathematical facts

- If space is covered, #partitions grows exponentially
  - But usually there are not “exponentially many” points
  - Most partitions will be almost empty

- Average distances grows steadily

- Consider a 1-NN query
  - 1-NN queries search a hypersphere, but partitions are hypercubes
  - The larger d, the smaller the fraction of space a hypersphere of radius 0.5 fills within a hypercube of edge length 1
  - The larger d, the more partitions one has to search to find neighboring points – the space is empty, everything is far away