Content of this Lecture

- Introduction
- Partitioned Hashing
- Grid Files
- \texttt{kdb Trees}
- R Trees
- Example: Nearest neighbor image search
kd Tree

- Grid file disadvantages
  - All hyperregions of the d-dimensional space are eventually split at the same 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 is a main memory data structure
General Idea

- Binary, rooted tree
- Inner nodes define splits (dimension / value)
- Dimensions need not be statically assigned to levels of the tree
- Leaves: Points+TIDs
- Each leave represents d-dimensional convex hypercube with m border planes (m≤d)
The Brick Wall

The diagram illustrates the conditions for selecting bricks based on their dimensions. The conditions are:

- If $x < 3$, then the selection criteria are:
  - $y < 1$ for brick $(2,0)$
  - $y \geq 3$ for brick $(0,4)$ and $(1,1)$

- If $x \geq 3$, then the selection criteria are:
  - $y < 7$ for brick $(4,9)$
  - $x \geq 5$ for brick $(4,6)$ and $(3,3)$

- If $x < 5$, then the selection criteria are:
  - $y < 5$ for brick $(3,1)$

- If $x \geq 5$, then the selection criteria are:
  - $y \geq 2$ for brick $(6,4)$

The diagram uses a tree structure to represent these conditions, with each node indicating a condition and the branches leading to the corresponding bricks.
Local Adaptation
Search Operations

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

• **Exact point search**
  - In each inner node, **decide upon direction** based on split condition
  - Search inside leaf

• **Partial match query**
  - If dimension of condition in inner node is part of the query - proceed as for exact match
  - Otherwise, follow **all children in parallel** (multiple search paths)

• **Range query**
  - Follow **all children** matching the range conditions

• **Nearest Neighbor**
  - Chose likely “close-enough” range and perform range query
  - If no success, **iteratively broaden range**
kd-Tree Insertion

• Search leaf block; if space available – done
• Otherwise, chose split (dimension + position) for this block
  - This is a local decision, valid for subtree of this node
  - Option: Use each dimension in turn (very robust)
  - Option: Consider current points and split in two sets of approximately equal size
  - Option: Consider known distributions of values in different dimensions
  - Finding “optimal” split points is expensive for high dimensional data (point set needs to be sorted in each dimension) – use heuristics
  - Wrong decisions in early splits may lead to tree degradation
  - But we don’t know which points will be inserted in future
    • Use knowledge on attribute value distributions
Deletion

• Search leaf block and delete point

• If block becomes (almost) empty
  - Leave it – bad fill degree
  - Merge with neighbor leaf (if existing)
    • Two leaves and one parent node are replaces 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
  - Balance with neighborhood
    • Also considering maximal depth of leaves

• kd trees are not balanced

• There is no guaranteed fill degree in blocks
kd Trees on Block Storage – Naive Solution

- Store each inner node in one block
  - Inner blocks are essentially empty
  - As trees are high (at least $\log_2(n)$), every descend requires many IO
  - Since tree is not balanced, worst case may approach $O(n)$ IO
Better IO: Fill Inner Blocks

- **Option 1: Multiway branching**
  - Inner node splits a dimension at many scales
  - When leaf overflows, insert new split into parent
  - When leaf underflows, merge and remove split from parent
  - Still not balanced, no guaranteed fill degree
kdb trees

• Option 2: Maps many inner nodes to few inner blocks
  – Inner nodes still have two children (mostly in the same block)
  – Inner blocks have many children
    • Roots of kd trees in other blocks

• Operations
  – Searching: As with kd trees
    • But better IO complexity
  – Insertion/Deletion
    • Tree can be balanced similar to B+ Tree
Another View

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

- \( d=3, \ n=1\times10^9, \) block size 4096, \( |\text{point}|=9, \ |b-\text{ptr}|=10 \)
  - We need \( \sim2.2M \) leaf blocks

- **Composite B+ index**
  - Inner blocks store 108-215 pointers; assume optimal package
  - We need 3 levels
    - \( 2^{\text{nd}} \) level has 215 blocks and 46,000 pointers
    - \( 3^{\text{rd}} \) 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: \( \sim1250 \) IO
Visualization
Example: Partial Box Query

- Box query on 2nd and 3rd dimensions only, asking for a 5% range in each dimension
  - 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 = 24K data blocks
  - Altogether: 26,000 I/O

- 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 kd tree** will have \( \sim 22 \) levels
  - \( \sim 455 \) points in one block (assume optimal packaging)
  - We need to address \( \frac{1E9}{455} \sim 2^{21} \) blocks

- **Consider** \( 128 = 2^7 \) kb-tree nodes in one kdb-block
  - Rough estimate; we need to store 1 dim indicator, 1 split value, and 2 b-ptr for each inner node, but most b-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 2\(^{nd}\) level blocks holding levels 8-14 of kd tree
  - There are \( \sim 16000 \) 3\(^{rd}\) level blocks, each addressing 128 data blocks
Space Covered

- 1st block evenly *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 2nd level block
  - Very likely, we need to look at only one 3rd level block
  - Assume we need to load all therein addressed 128 data blocks
  - Altogether: $1+1+1+128 = 131$ 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 ~2 splits in each dimension
    - Two times 2 splits, one time three splits
    - Assume we miss the dimension with 3 splits
  - Hence, in ~4 of 7 splits we know where we need to go, in ~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 3rd level blocks = 8*8*8 data blocks
  - Altogether: 1+8+64+512 ~580 IO
    - Compare to 3100 for composite index
Beware

- We made many assumptions
- Uniform distribution is rare
- Trees are rarely balanced
- Optimal packaging of points rare
- Performance can greatly vary due to # of dimensions, distribution of values, order of insertions and deletions, selectivity, split and merge policies, …
Conclusion

- **kdb trees can be balanced**
  - Similar method as for B+ trees
  - When splitting a leaf, a new node must be inserted into parent
  - Overflow may walk up to root
  - When inner nodes are split, splits must be propagated downward
    - As regions need to stay convex

- **kdb trees have problem with fill degree**
  - Many insertions/deletions lead to almost empty leaves
  - Index grows unnecessarily large
  - No guarantee for lowest fill degree as in B+ tree
Content of this Lecture

• Introduction
• Partitioned Hashing
• Grid Files
• kdb Trees
• R Trees
• Example: Nearest neighbor image search
General Idea

- Objects are hierarchically grouped into regions
- Regions may overlap
- Objects are represented only once
Example (k=3)

- We need overlapping regions
  - For instance, if all rectangles overlap
  - No split possible which creates disjoint sets of objects
R-Trees


- Can store geometric objects (with size) as well as points
  - For geometric objects, use 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

- Many variations (see literature)
R Tree versus kd Tree
Concepts

• Inner nodes consist of a set of \textit{d-dimensional regions}
  • Or cube or hypercube
• Regions are hierarchically contained in larger regions
• Each region is represented by a subtree or a leaf
• The \textit{region border} is the MBB of all objects in its subtree
  • Inner node: MBB of all child regions
• Leaf blocks: All objects contained in the respective region
• Regions in one level may \textit{overlap}
  • Also regions within a node may overlap
• Regions of a level need not cover the space completely
Searching

- Typical query: Find all objects (rectangles) overlapping with a given query rectangle
- In each node, **intersect with all regions**
- More than one region might have non-empty overlap
- **All must be considered**
  - No $O(\log(n))$ search complexity
Inserting an Object

• In each node, consider candidate regions
  - Regions may overlap the object completely, partly, or not
  - Within a node, no region may overlap with the object
  - More than one region with complete overlap?
    • Chose one (smallest?) and descend
  - None with complete, but several with partial overlap?
    • Chose one (largest overlap?) and descend
  - No overlapping region at all?
    • Chose 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
One State
Example: Searching

No overlap in child regions (only in MBB) – stop search
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)
Choosing Splits of Overflow Blocks

• Option 1: Avoid overlaps, cover large space
  - 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: Allow overlaps, 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
Block Splits

• Whatever strategy we chose
  - Consider a block with n objects
  - There are $2^n - 2$ possibilities to partition this block into two
  - In multi-dimensional spaces, there is no simple sorting
  - Use heuristics instead of optimal solution

• R* tree
  - Chose as criterion combination of sum of covered spaces, space of intersection, and sum of girt ("Umfang")
  - Use heuristic for concrete decision
  - Established strategy
Multidimensional Data Structures Wrap-Up

• We only scratched the surface (balancing, deletions)
• Many other: X tree, hb-tree, R+ tree, UB tree, …
  – Store objects more than once; other than rectangular shapes; map
cordinates into integers; …
• Curse of dimensionality
  – The more dimensions, the more difficult
    • Balancing the tree, finding MBBs, split decisions, etc.
  – All MDIS at some point degenerate
    • Exploding size of directories, empty kdb-trees, all regions overlap, …
  – Often, linear scanning of objects is quicker
    • Or: Compute lower-dimensional, relationship-preserving approximations of objects and filter on those
Content of this Lecture

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

**Example: Nearest neighbor object search**
- Material partly from A Müller, 2003
- Korn, Sidiropoulos, Faloutsos, Siegel, Protopapas (1996): Fast Nearest Neighbor Search in Medical Image Databases, VLDB.
Similar Objects in Images
Two Tasks

• A **distance function**: How similar are 2D objects?
  - Shape, size, rotation, borders, ...

• Similarity search: **Fast algorithm** to all similar / the most similar objects in a database of objects
  - Brute force: Compare against all objects

• Trick: Fast, **iterative filtering** of candidates
Distance Function

• Requirements
  - Should be insensitive to rotation
  - Should consider overall shape (macro-scale) as well as structure of the surface (micro-scale)

• One option: Mathematical morphology
  - Idea: Use brushes to fill / surround the objects
  - Opening: Area covered when filling object with brush
  - Closing: Area covered when surrounding object with brush
  - Using brushes with different thickness gives different areas and thus different approximations
Examples

original shape

after opening

after closing

[Diagram showing different shapes and their transformations]
Distance Function

- Overlay objects $o_1$ and $o_2$
  - Align centers of mass
  - Rotate until maximal overlap
- Assume we use $n$ brushes $B_1, \ldots, B_n$
- For each brush $B_i$, compute
  - $O_{1i}/C_{1i}$: Area under opening / closing of $o_1$ with $B_i$
  - $O_{2i}/C_{2i}$: Area under opening / closing of $o_2$ with $B_i$
- Define
  $$\text{dist}_i(o_1, o_2) = \max\left(\frac{O_{1i} \cap O_{2i}}{O_{1i} \cup O_{2i}}, \frac{C_{1i} \cap C_{2i}}{C_{1i} \cup C_{2i}}\right)$$
- Define $\text{dist}(o_1, o_2) = \max(\text{dist}_1(o_1, o_2), \ldots, \text{dist}_n(o_1, o_2))$
Evaluation

- **Very precise** method
- Adaptable by varying $n$ / thickness of brushes
- Highly complex -> very slow
  - Frequent computation of spatial overlaps between irregular shapes
- Cannot be used to search against thousands of objects
- Idea
  - Find a distance function $d_{\text{app}}$ such that $d_{\text{app}}(o_1, o_2) \leq \text{dist}(o_1, o_2)$
  - If we have a max distance $t$ and find $d_{\text{app}}(o_1, o_2) > t$, we also know that $\text{dist}(o_1, o_2) > t$
  - Function $d_{\text{app}}$ allows pruning
  - Only helps if $d_{\text{app}}(o_1, o_2)$ is (a) computable quickly and (b) approximates $\text{dist}$ well
Intuition

Distanz

$\text{Filter}$

$\text{Exakt}$

$\text{Näherung}$

$t = 5$

$k$

Spectrum Function

- Consider values $O_{11}, O_{21}, \ldots O_{n1}$ (and $C\ldots$)
- Compute spectrum of $o_1$: Vector with differences $O_{11}-O_{21}, O_{21}-O_{31}, \ldots$
- Distance between two spectra is a lower bound for true distance function $\text{dist}$

\[
\text{Spectrum} = [...; 0.5; 0.8; 1.5; 5]^T
\]
Algorithm

- Spectra can be pre-computed and indexed
- Use nearest neighbor search in multidimensional index
- Optimization: Use iterative procedure
  - Start with large value $t$
  - Find first objects within range $t$ using approx search
  - Compute real distance and use as new $t$
  - Iteratively prunes search space
Effect

- Full database scan: \( \sim 14 \text{h} \)
- Iterative pruning

Graph showing 10-nearest neighbors with response time (seconds) vs. db size (N).