Datenbanksysteme II: Multidimensional Index Structures 2

Ulf Leser

Content of this Lecture

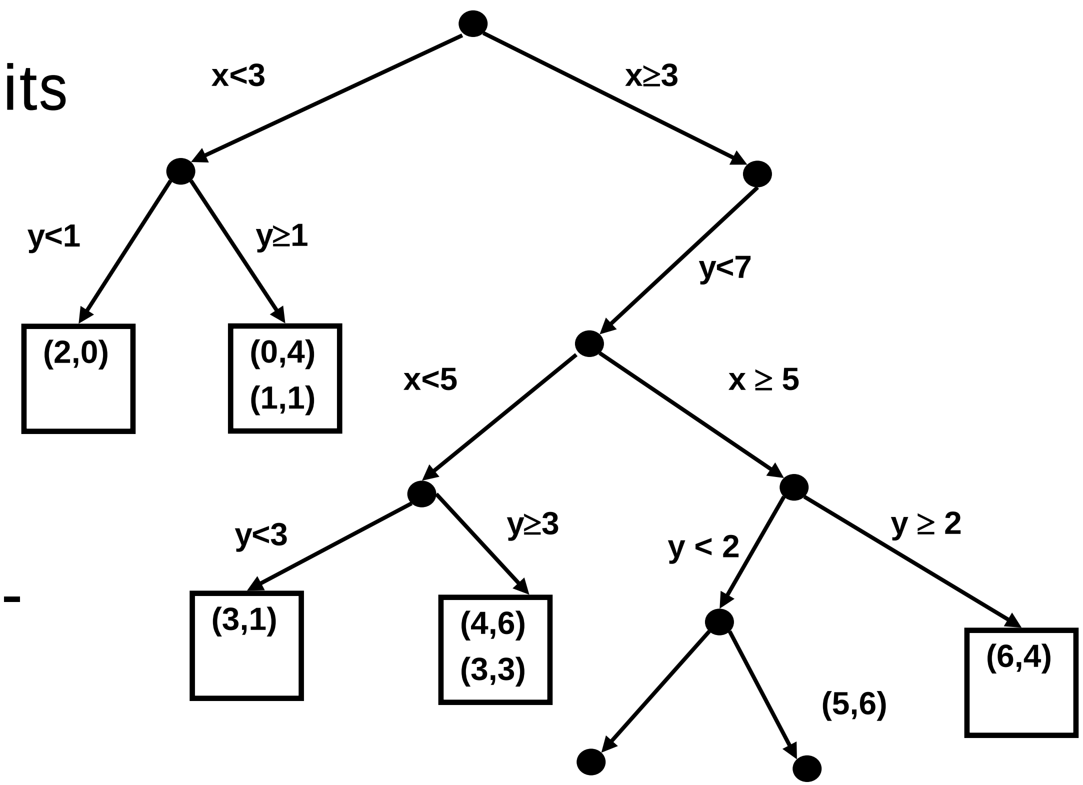
- Introduction
- Partitioned Hashing
- Grid Files
- kdb Trees
  - kd Tree
  - kdb Tree
- R Trees
- Example: Nearest neighbor image search

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
  - Bentley: Multidimensional Binary Search Trees Used for Associative Searching. CACM, 1975.
  - 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

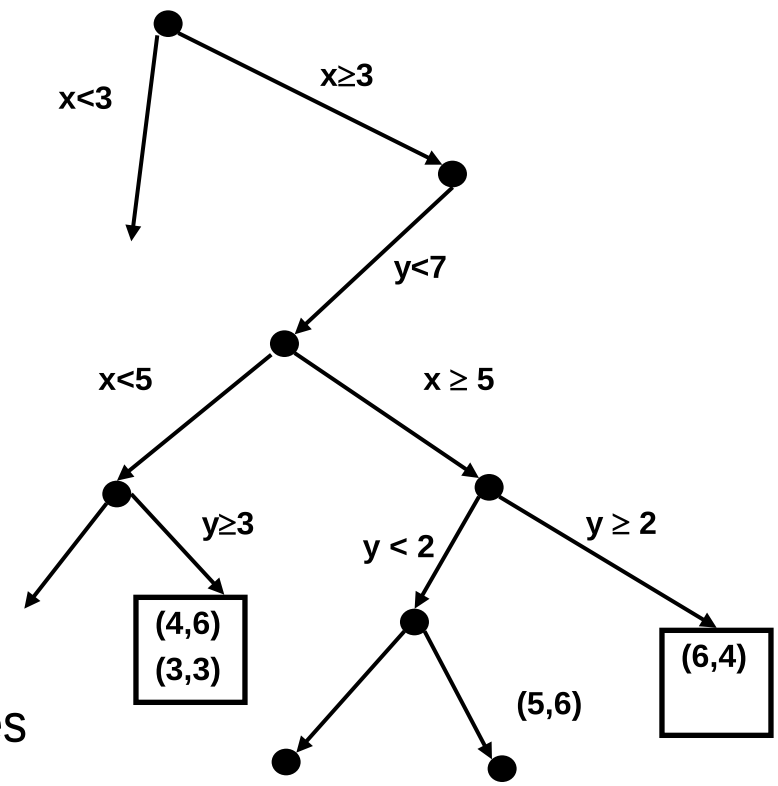
General Idea

- Binary, rooted tree
- Inner nodes define splits (dimension / value)
- Dimensions need not be statically assigned to levels of the tree
- Leaves: Points+TIDs
- Each leaf represents d-dimensional convex hypercube with m border planes (m≤2d)

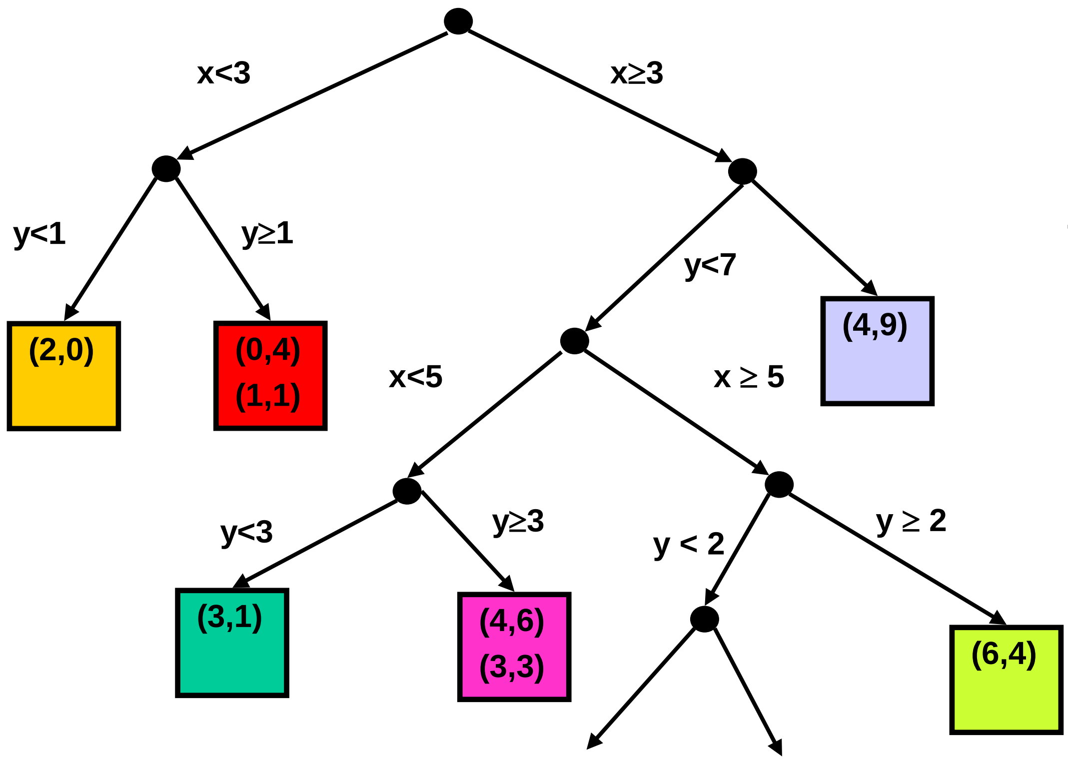


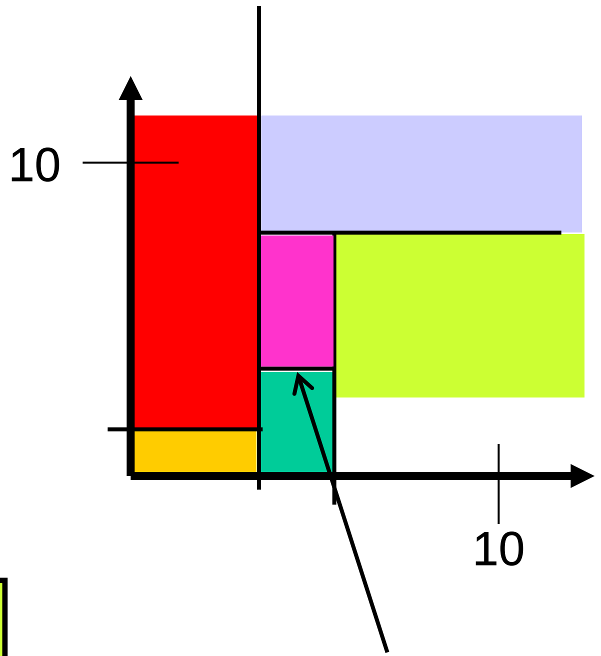
Blocks and Points

- Keep everything in memory
  - Leaves are singular points
- Keep tree in memory and blocks on disk
  - Leaves contain many points
- Store everything on disk
  - k-DB Tree: Special layout for tree
- On modern hardware
  - Block size – level 1/2/3 cache
  - Random mem access in inner tree
  - But larger leaves create smaller trees
  - Parallel search? SIMD? Tree layout?



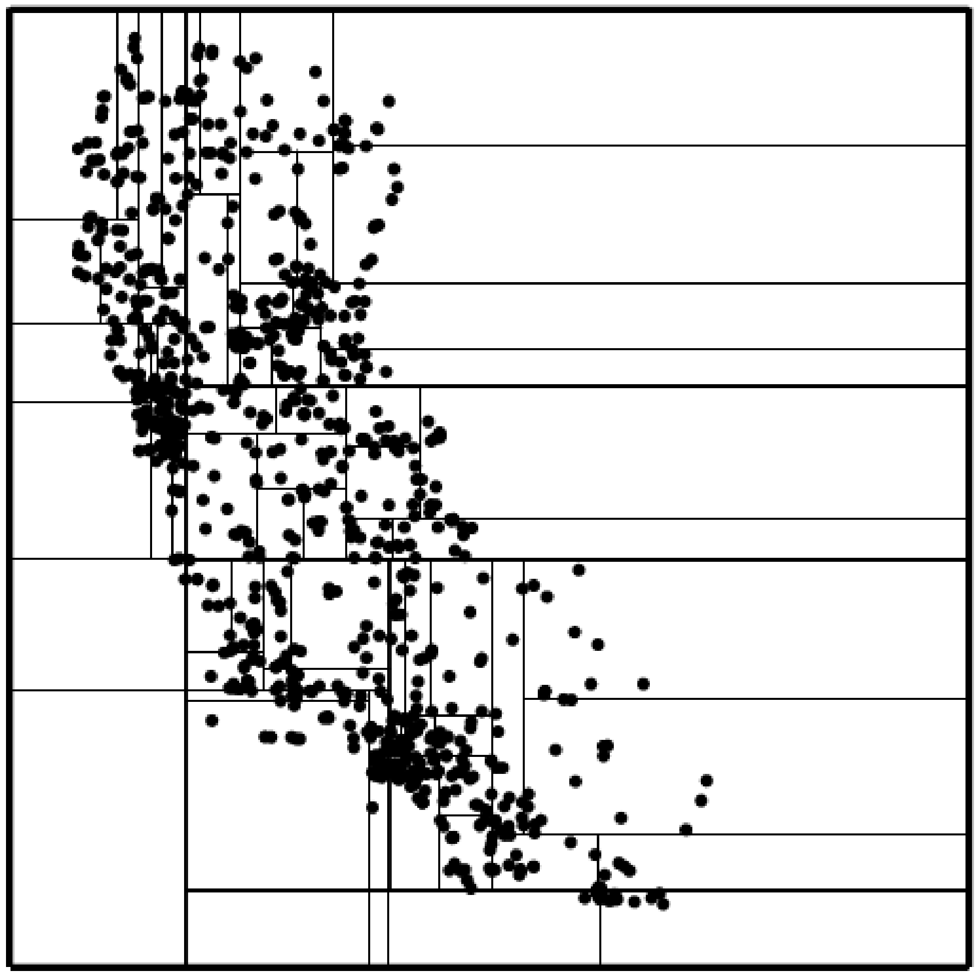
The Brick Wall





Splits are local
(for their subtree)

Local Adaptation



Search Operations

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

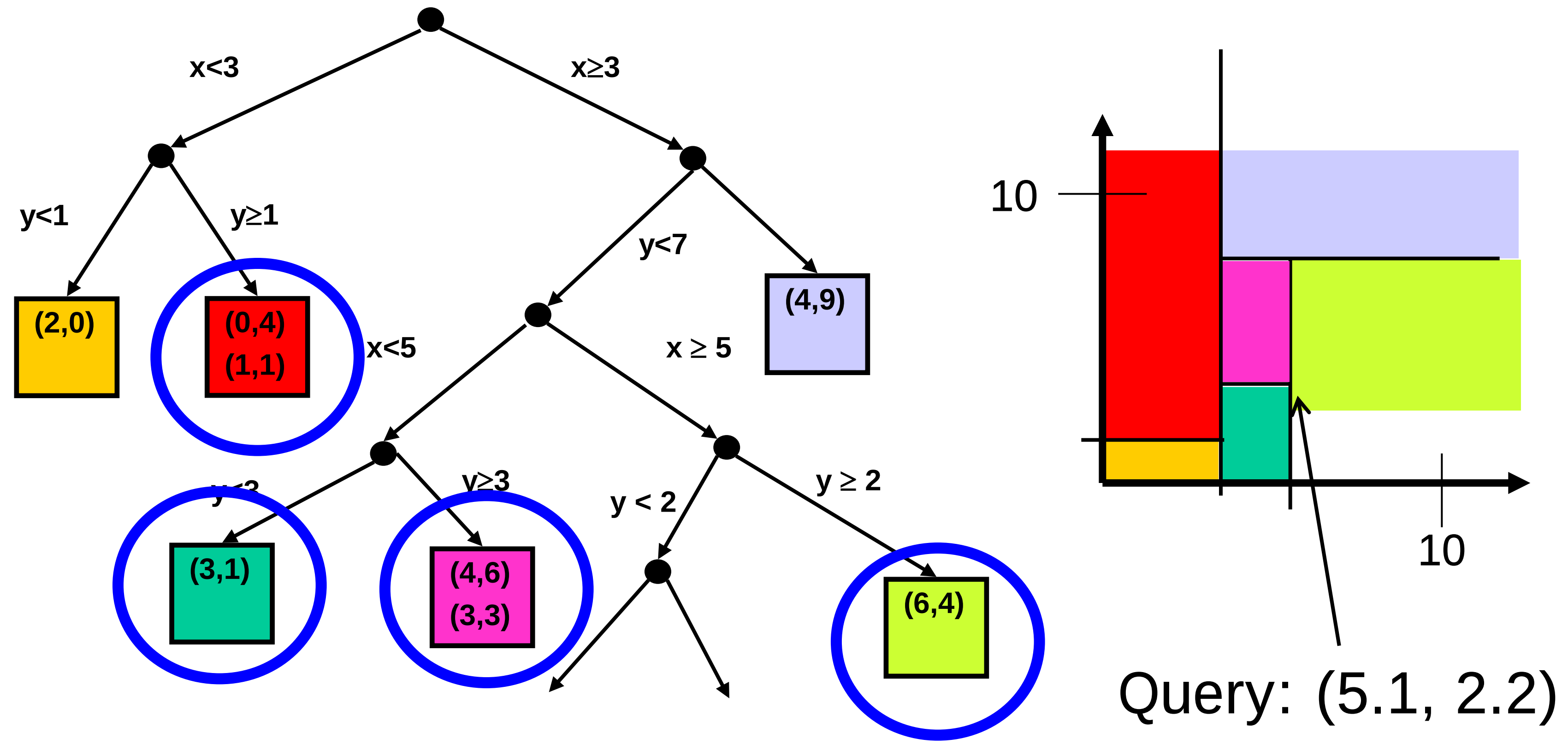
Search Operations

- 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 (multiple search paths)
- 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

The Brick Wall



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 and split region into two equally sized subspaces (very robust)
  - Option: Consider current points in leaf and split in two sets of approximately equal size
  - 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 have no guaranteed balance (~ depth)
- There is no guaranteed fill degree in blocks

Static kd Trees

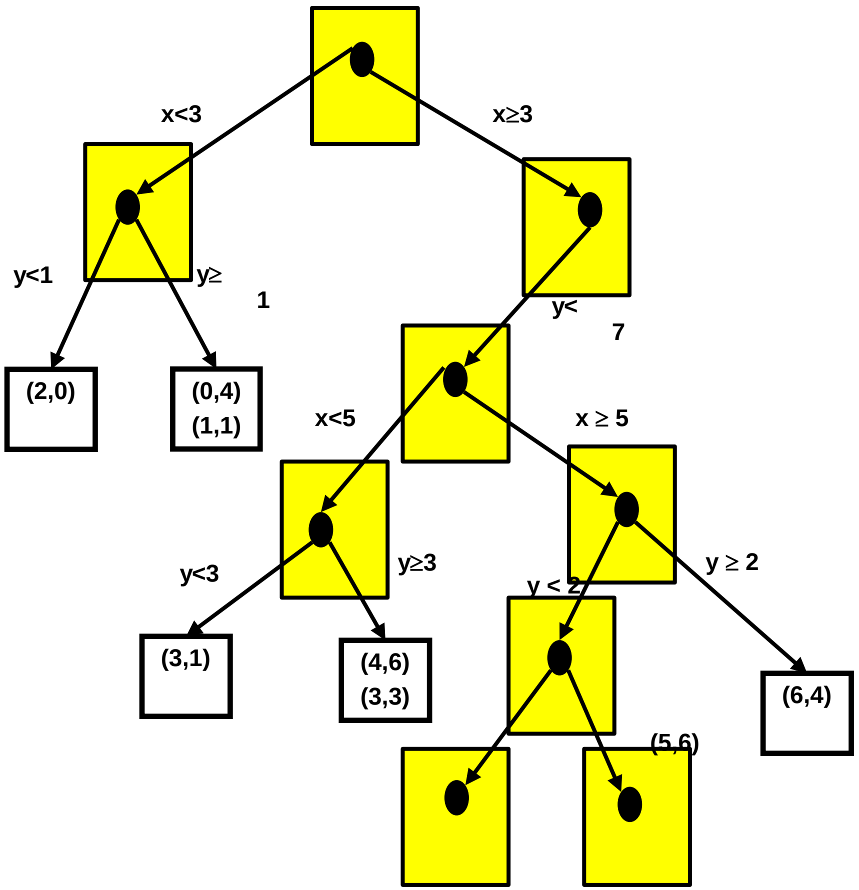
- Assume the set of points to be indexed is static and known
- Worst-case optimal kd Trees
  - Rotate through dimensions
    - Typically in order of variance – wide spread dimensions first
  - Sort remaining points and choose median as split point
  - Guarantees tree depth of O(log(n)) for point queries
  - But clustering of points not considered – bad similarity queries
    - Nearby points are not nearby in the tree
- K-means trees
  - Iterative k-means clustering of points
  - K: Tree width (fanout)
  - Faster similarity queries, tree depth not guaranteed
- n-Ary kd-Trees for exploiting SIMD instructions

Content of this Lecture

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

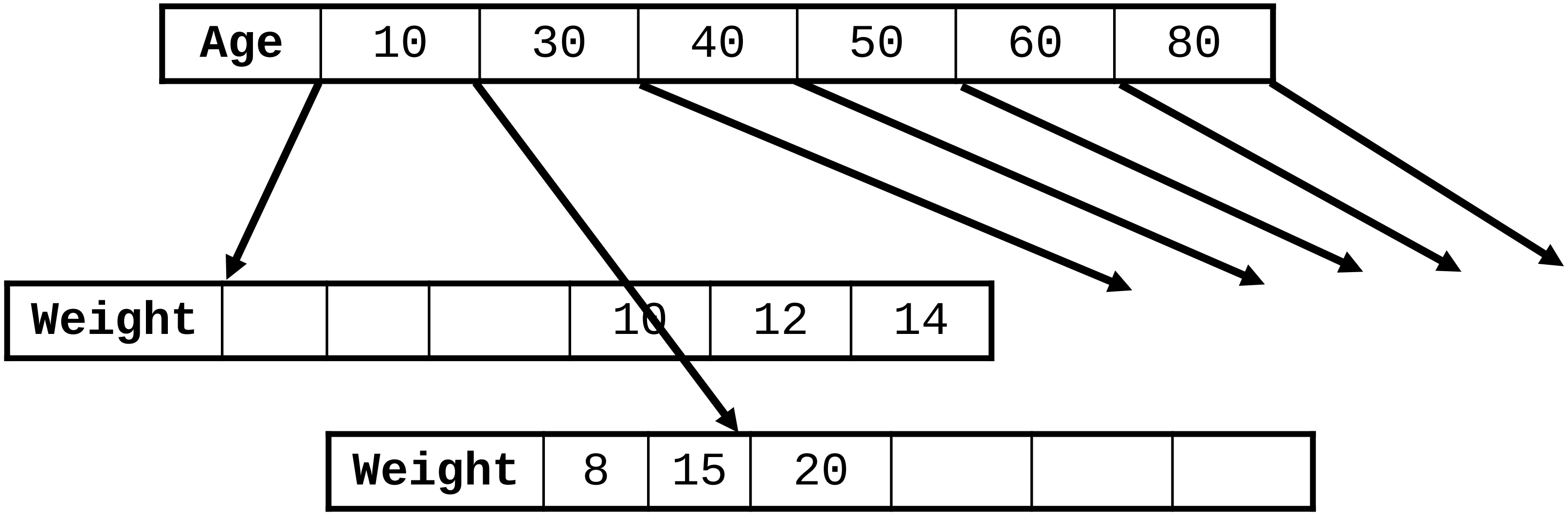
kd Trees on Secondary (Block) Storage – Naive Solution

- Store each inner node in one block
  - Inner blocks are essentially empty
  - As trees may degrade, every search requires many IO
  - Since tree is not balanced, worst case approaches O(n) IO



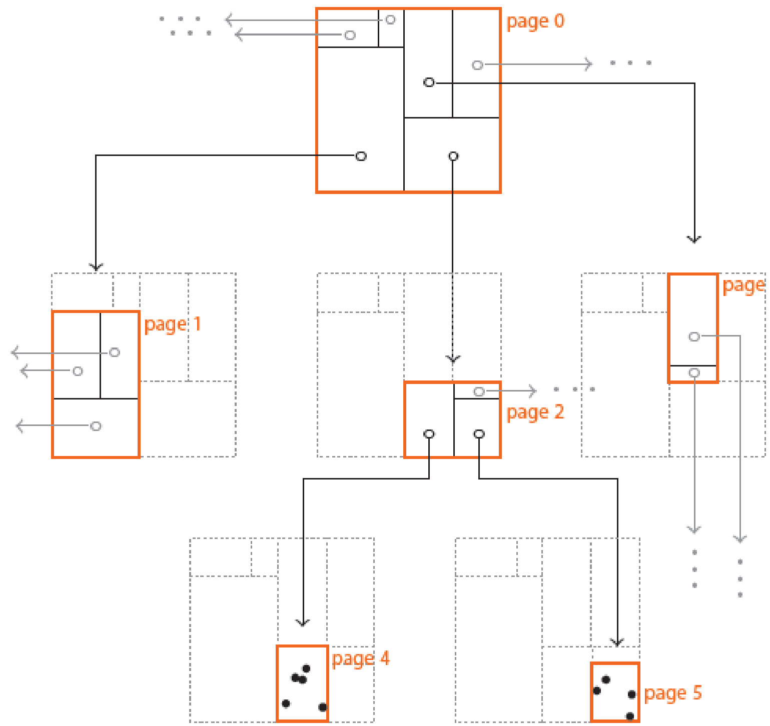
Better IO: Fill Inner Blocks

- Option 1: Build k-ary trees
  - 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



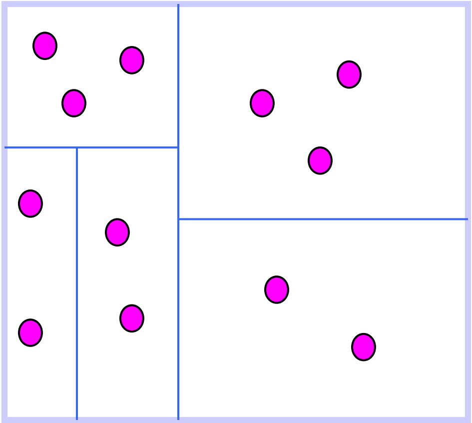
kdb trees

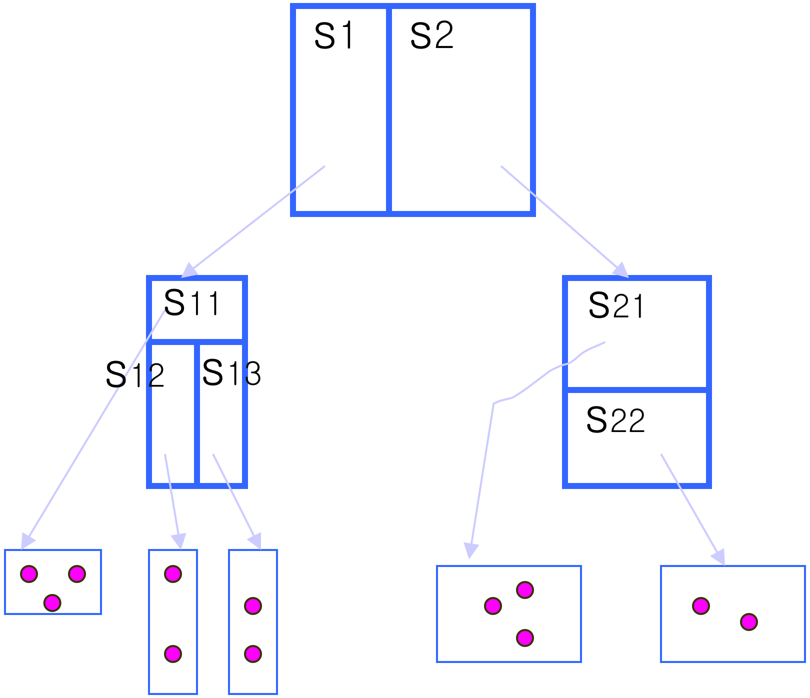
- Option 2: Map many inner nodes to a single blocks
  - Robinson: The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. SIGMOD 1981.
  - Inner nodes have two children (mostly in the same block)
  - Each block holds many inner nodes
  - Inner blocks have many children
    - Roots 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

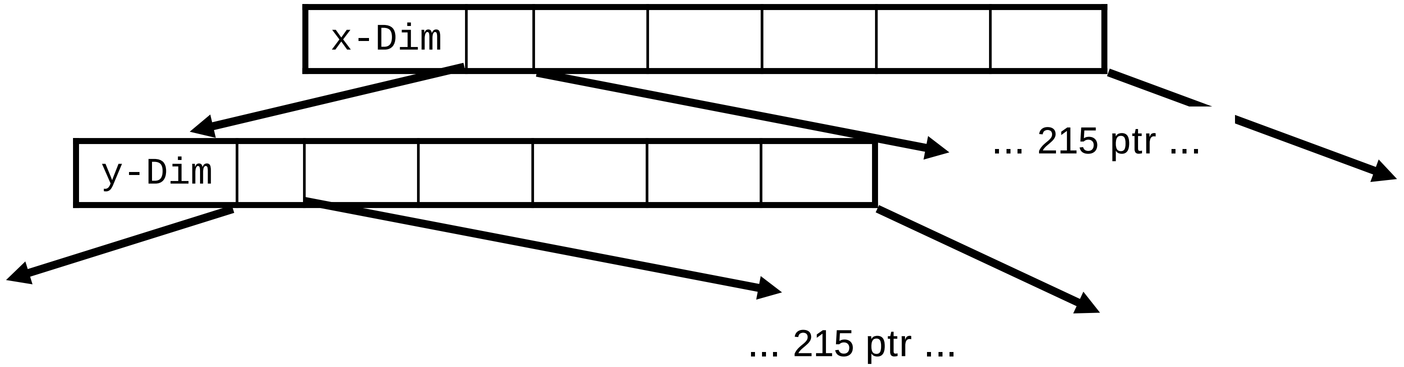




Example – Composite Index

- d=3, n=1E9, block size 4096, |point|=9, |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

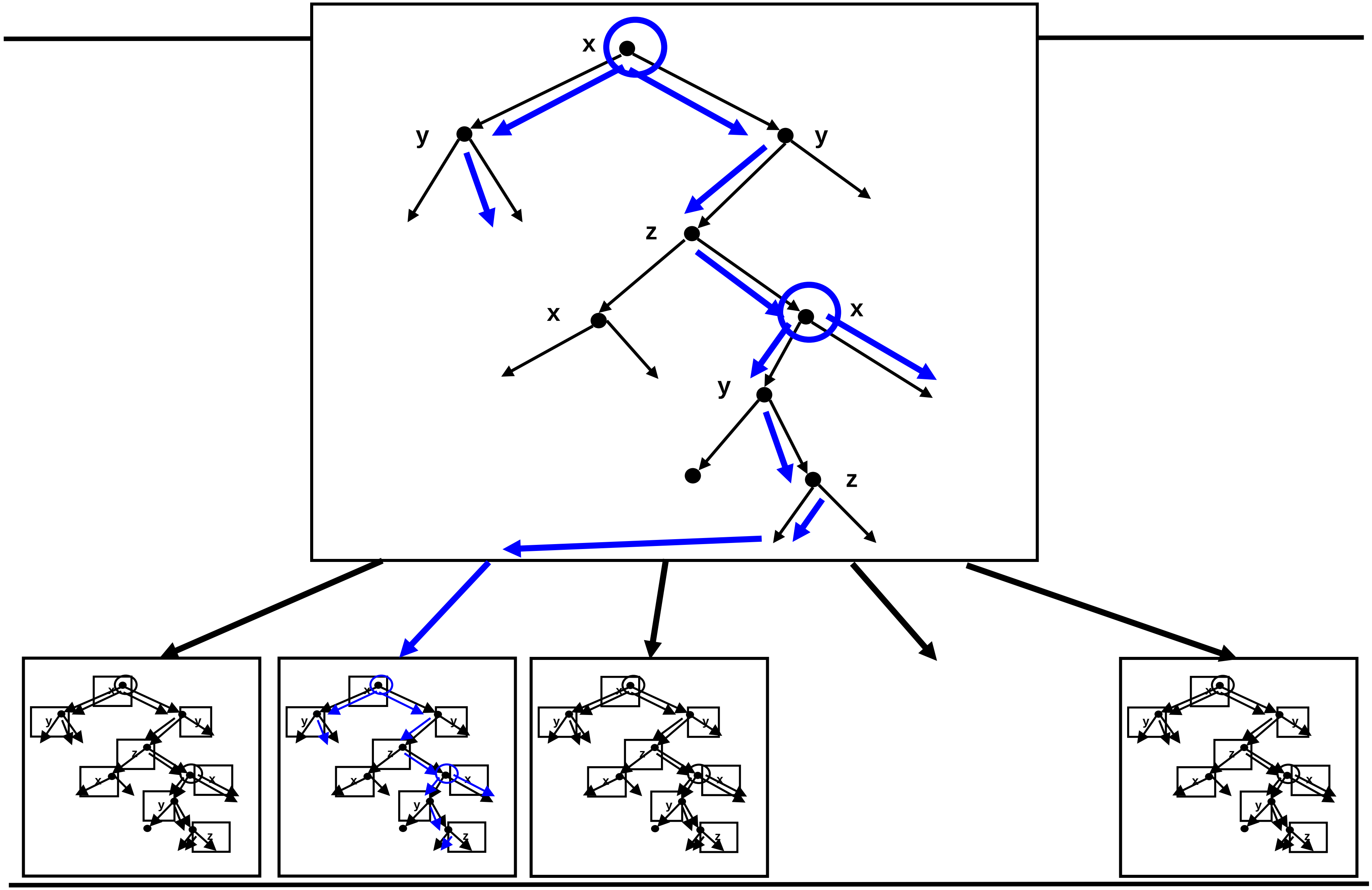
- Box query on 2nd and 3rd dimensions only, asking for a 5% range in both dimensions
  - We need to scan all 215 2nd level blocks
    - Each 2nd level block contains the 5% range of 1st dimension
  - For each, we read 5% of 3rd level blocks = 2300 blocks
  - For each, we read 5% of data blocks = ~25K 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 ~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
  - Not true in high dimensional spaces – everything becomes a border
    - See later: Curse of Dimensionality


x
y
y
z
x
x
y
z

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 3nd level blocks = 8*8*8 data blocks
  - Altogether: 1+8+64+512 ~580 IO
    - Compare to 3100 for composite index

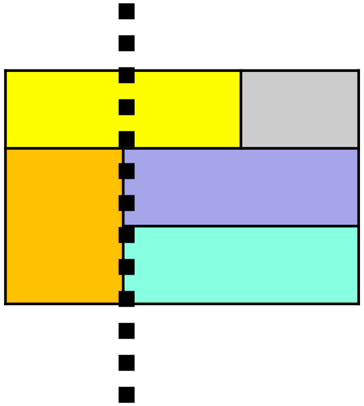
It‘s the Workload, st ...

- Advantages depend on expected queries

- Composite indexes are optimal if prefix of composite key is (heavily) constrained by the query
  - Comp-index also “partition” the space
  - Comp-index is similar to a kd-tree where in the first levels, only dimension X is used, then only dimension Y, ...
- MDIS are better if queries address neighboring points in many dimensions (box queries, neighborhood queries)
  - “Better” depends a lot on data and workload distribution
- Scanning is better when selectivity of queries is low

Balancing upon Insertions

- Similar method as for B+ trees
  - Search appropriate leaf
  - If leaf overflows, split
    - Chose dimension and scale; distribute points
    - 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 scale
  - Distribute nodes to the two new blocks
    - Split might have to be propagated downwards
    - “Default” split may lead to very bad fill degree
  - Propagate new pointers to parent



Conclusion

- Beware our simplifying assumptions
  - Uniform distribution
  - Optimal packaging of points at all levels
  - Query ranges contained in hypercubes
- kdb trees have problem with fill degree
  - Many insertions/deletions lead to almost empty leaves
  - Index grows unnecessary large
  - No guarantee for lowest fill degree as in B+ tree
- Nice idea, difficult to implement, rarely used in practice

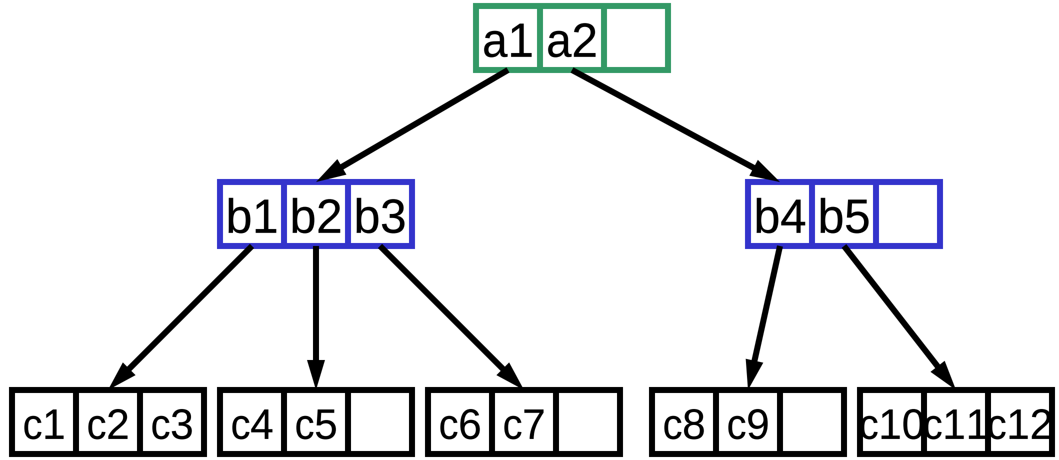
Content of this Lecture

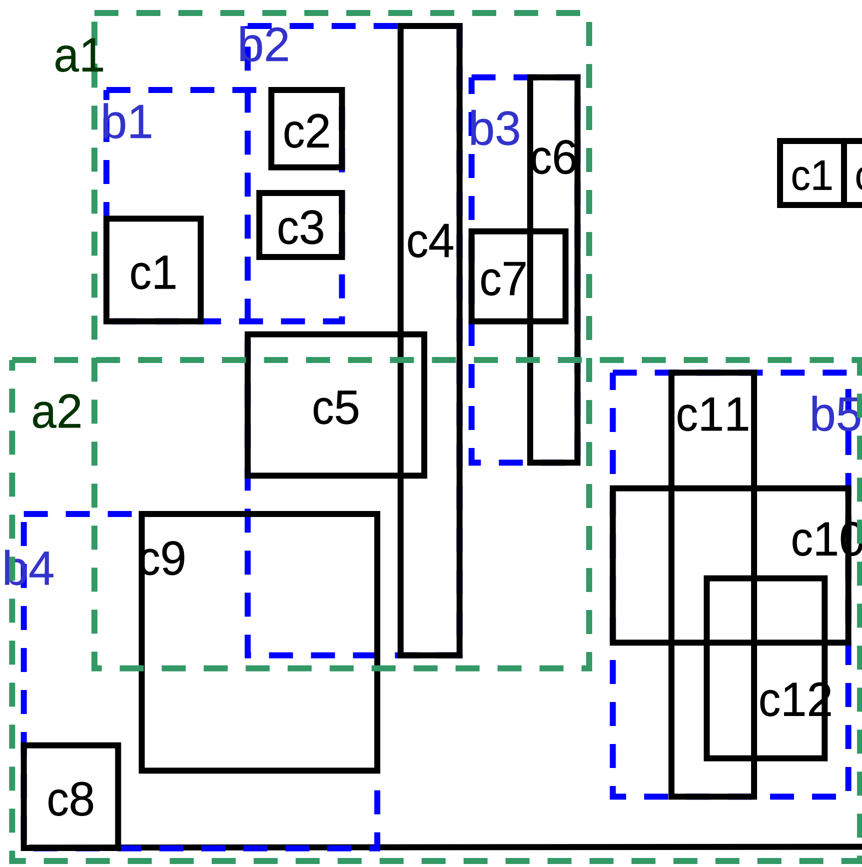
- Introduction
- Partitioned Hashing
- Grid Files
- kdb Trees
- R Trees
- Conclusions
- Example: Nearest neighbor image search

R-Trees

- Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD 1984.
- Can store geometric objects (with size) 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

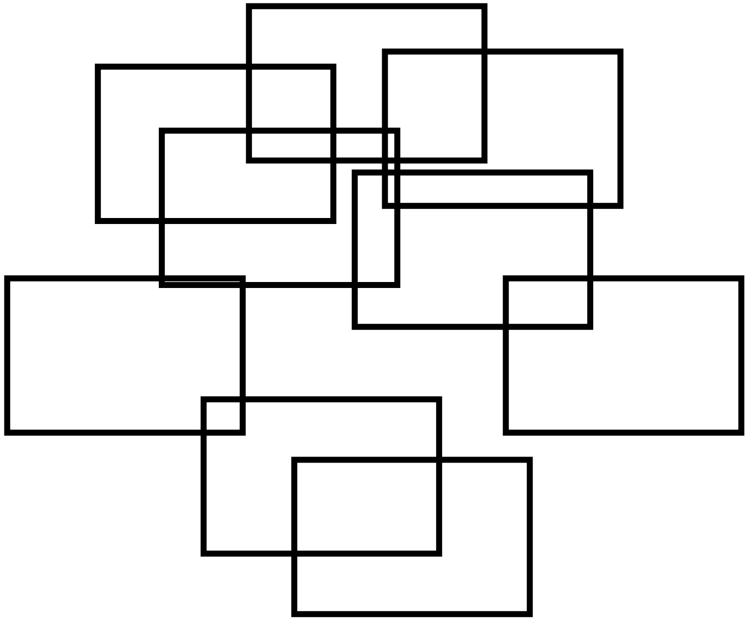




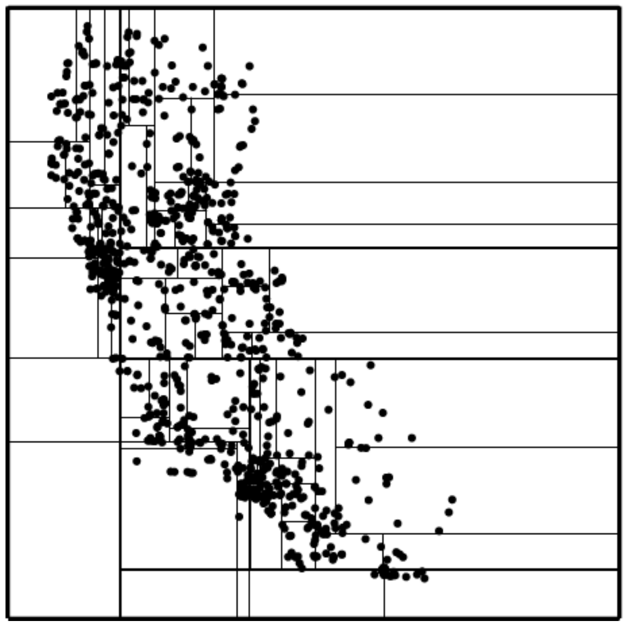
- Spatial objects
  - Represented by their minimal bounding box (MBB)
- Objects are hierarchically grouped into overlapping regions
- Objects are stored only once

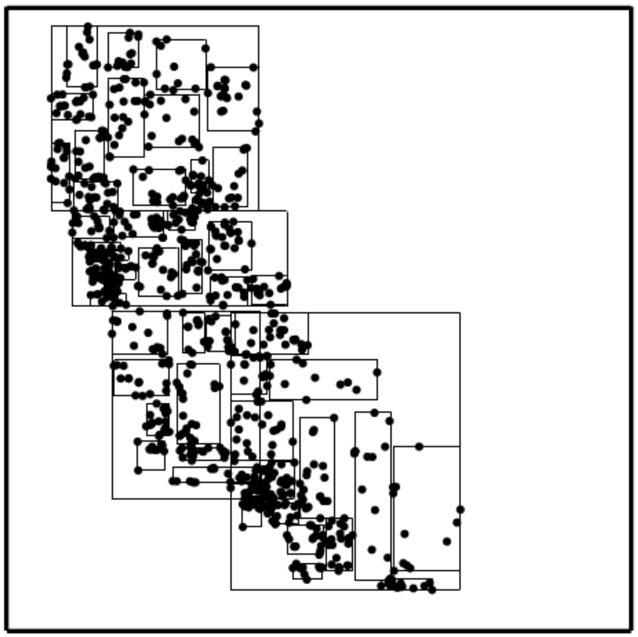
Motivation: Objects that are not points

- We need overlapping regions
  - For instance, if all MBBs overlap
  - No split possible which creates disjoints sets of objects
- Objects crossing a split
  - Store in only one (R-Tree)
    - Search must examine both
    - No redundant data
  - Store in both (R+-Tree)
    - Search may chose 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

Searching

- 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

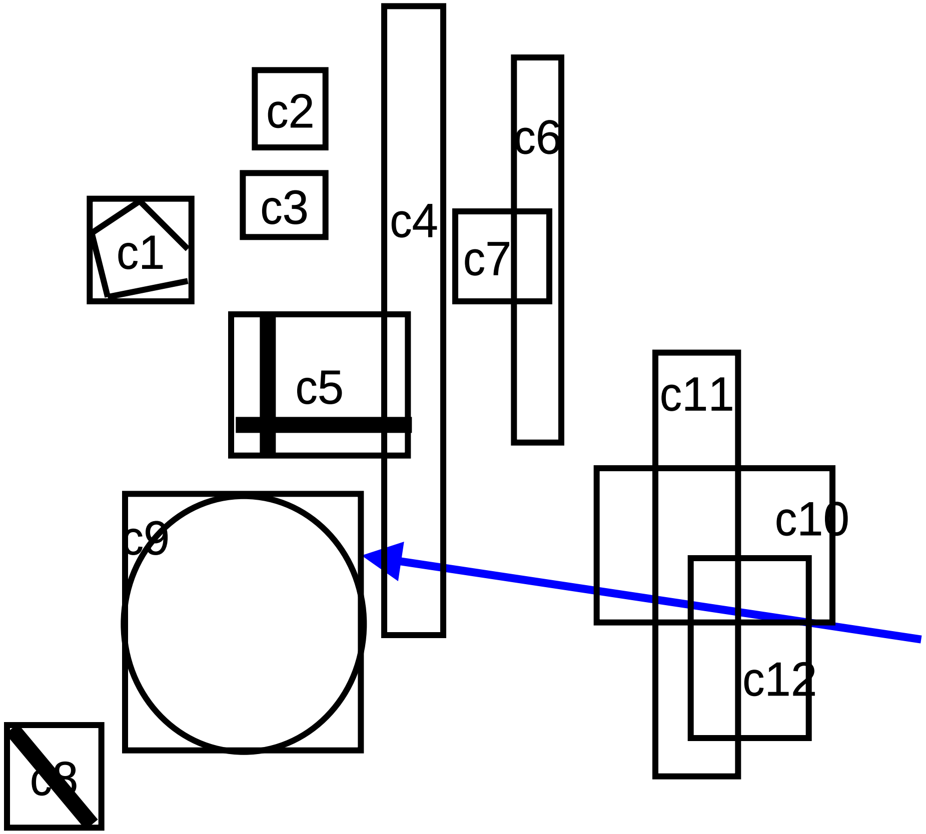
Inserting an Object

- 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
    - Chose one (smallest?) and descend
  - None with complete, but at least one 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

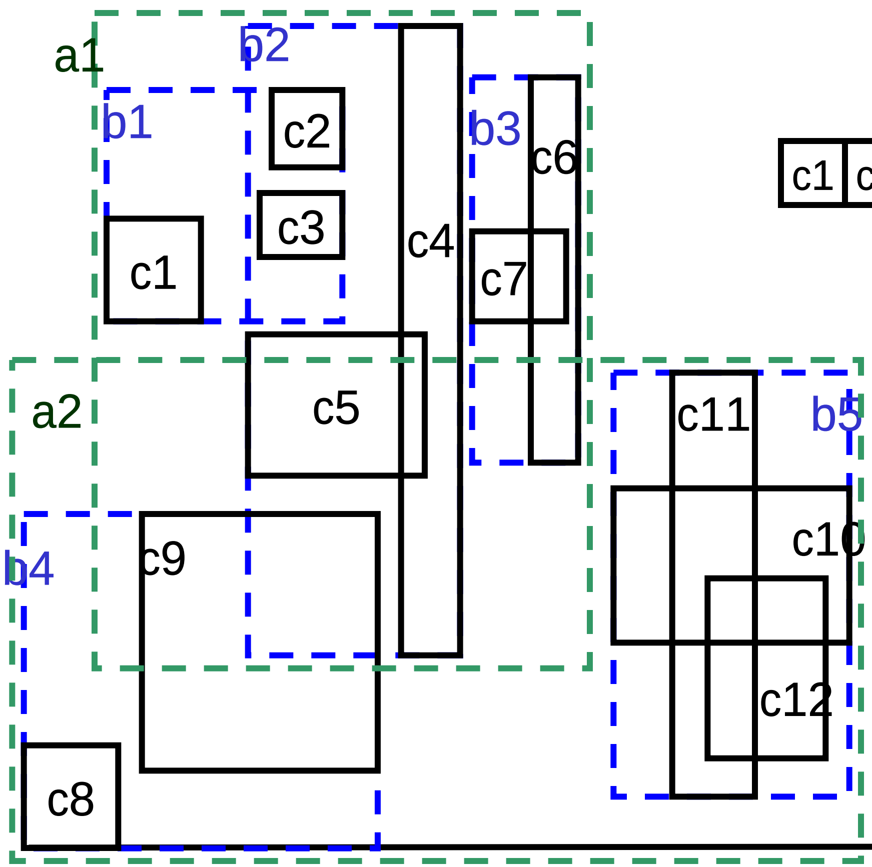
- If free space in leaf
  - Insert object and adapt MBB of leaf
  - Recursively adapt MBBs up the tree
  - This usually generates larger overlaps – search degrades
- If no free space in leaf
  - Split block in two regions
  - Compute MBBs
  - Adapt parent node: One more child, changed MBBs
  - May affect MBB of higher regions and/or incur overflows at high regions – ascend recursively

Example (from Donald Kossmann)

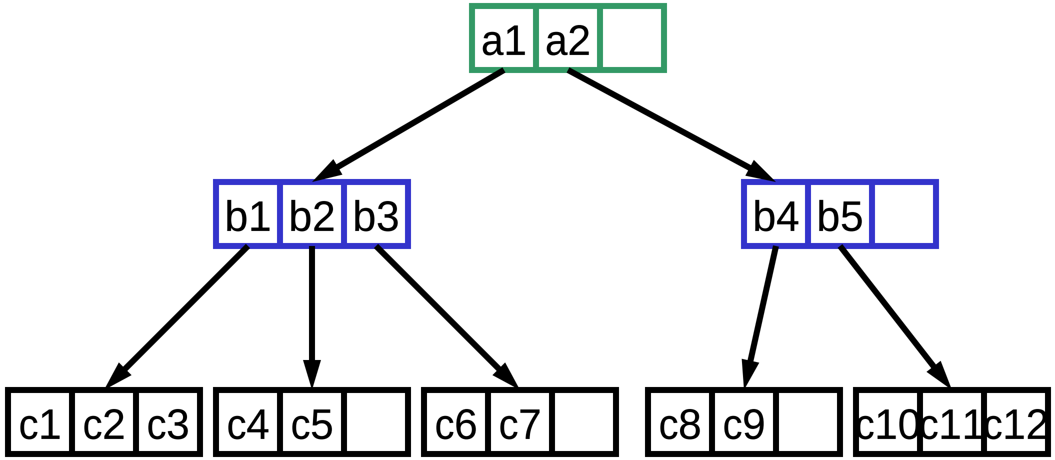


Compute MBBs for all non-rectangular objects

One State

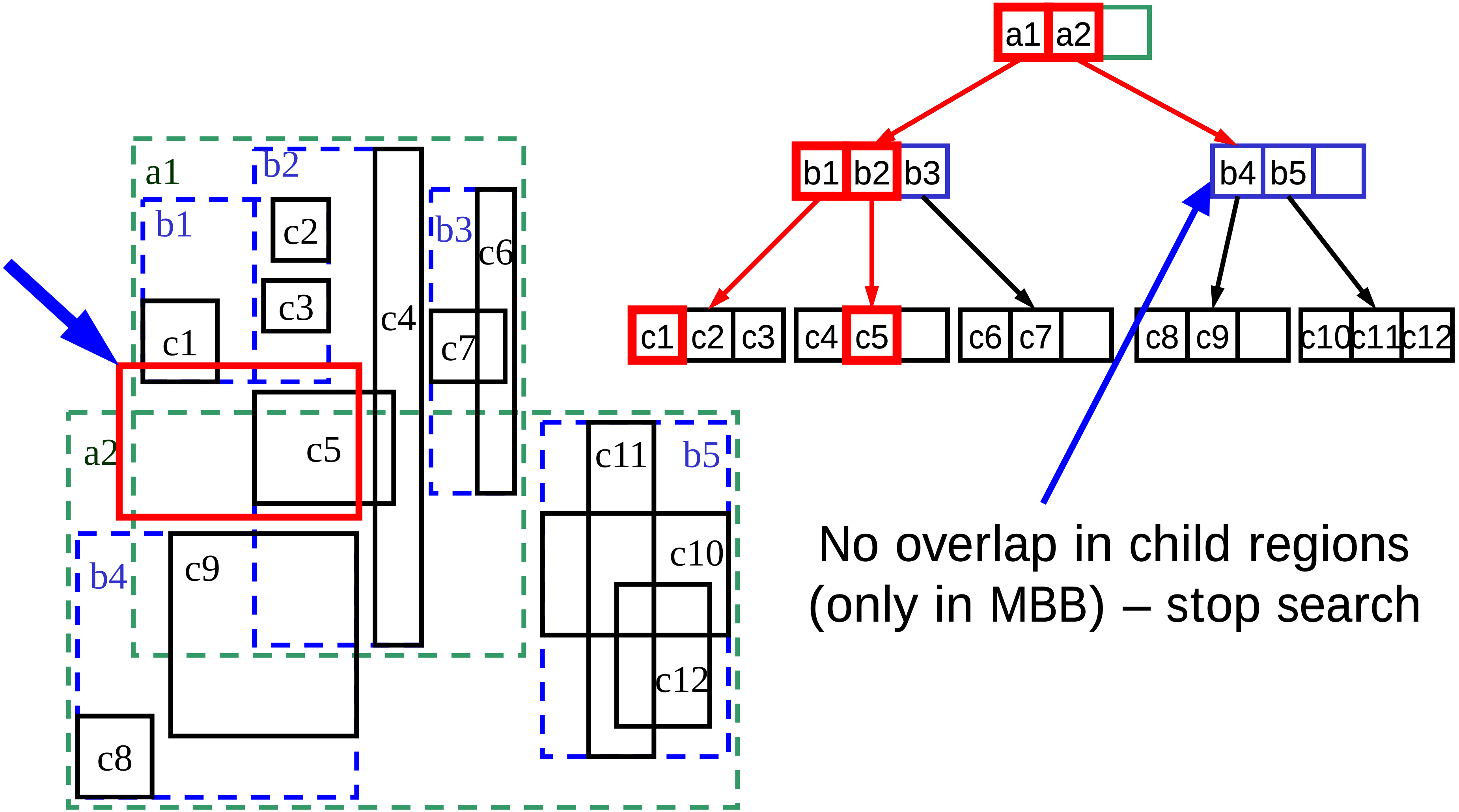


a1
a2
b1
b2
b3
b4
b5
c1
c2
c3
c4
c5
c6
c7
c8
c9
c10
c11
c12



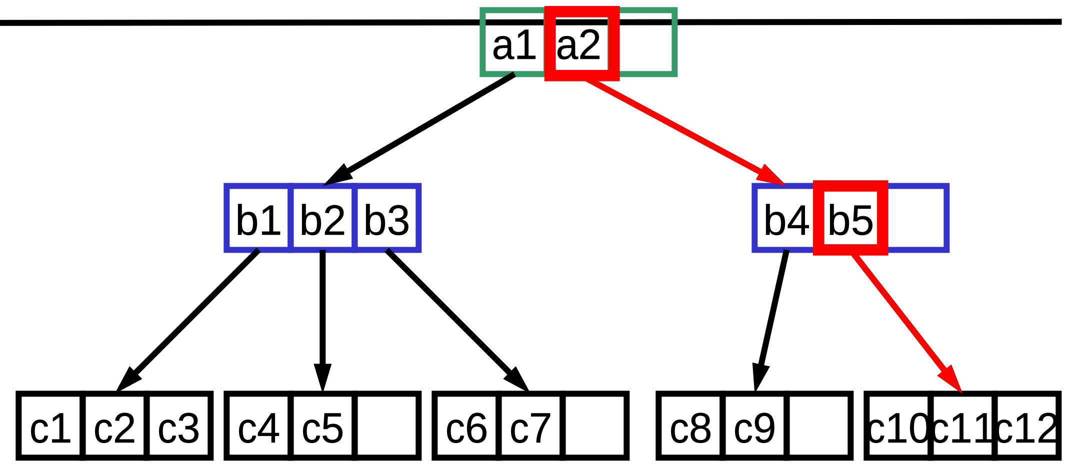
a1 a2
b1 b2 b3
b4 b5
c1 c2 c3
c4 c5
c6 c7
c8 c9
c10 c11 c12

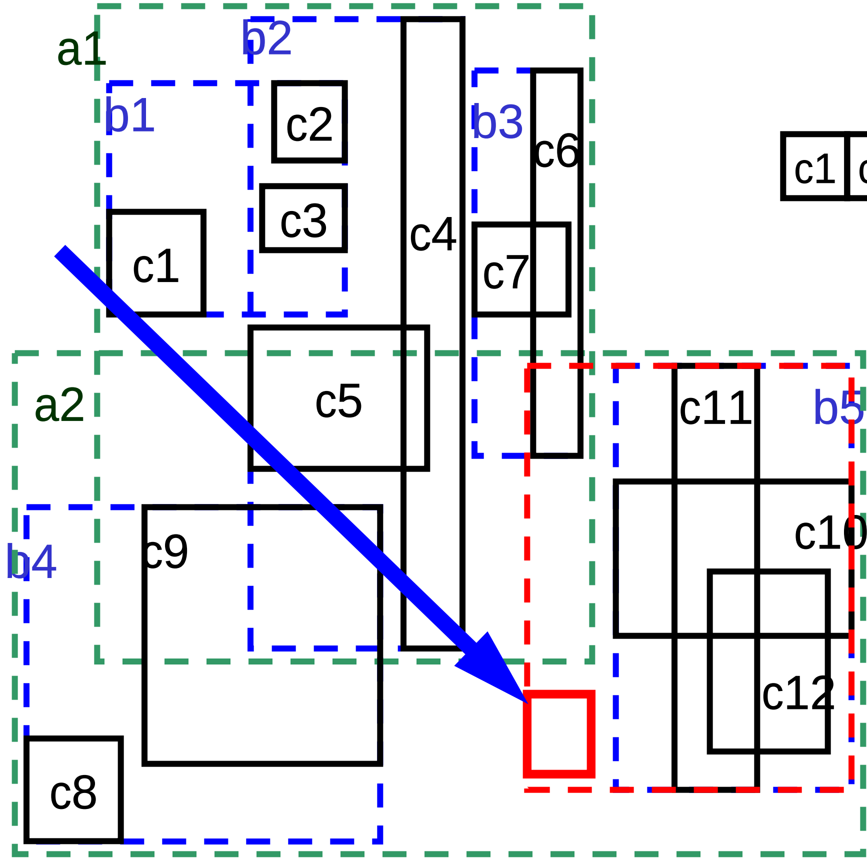
Example: Searching



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

Example: Insertion, Search Phase

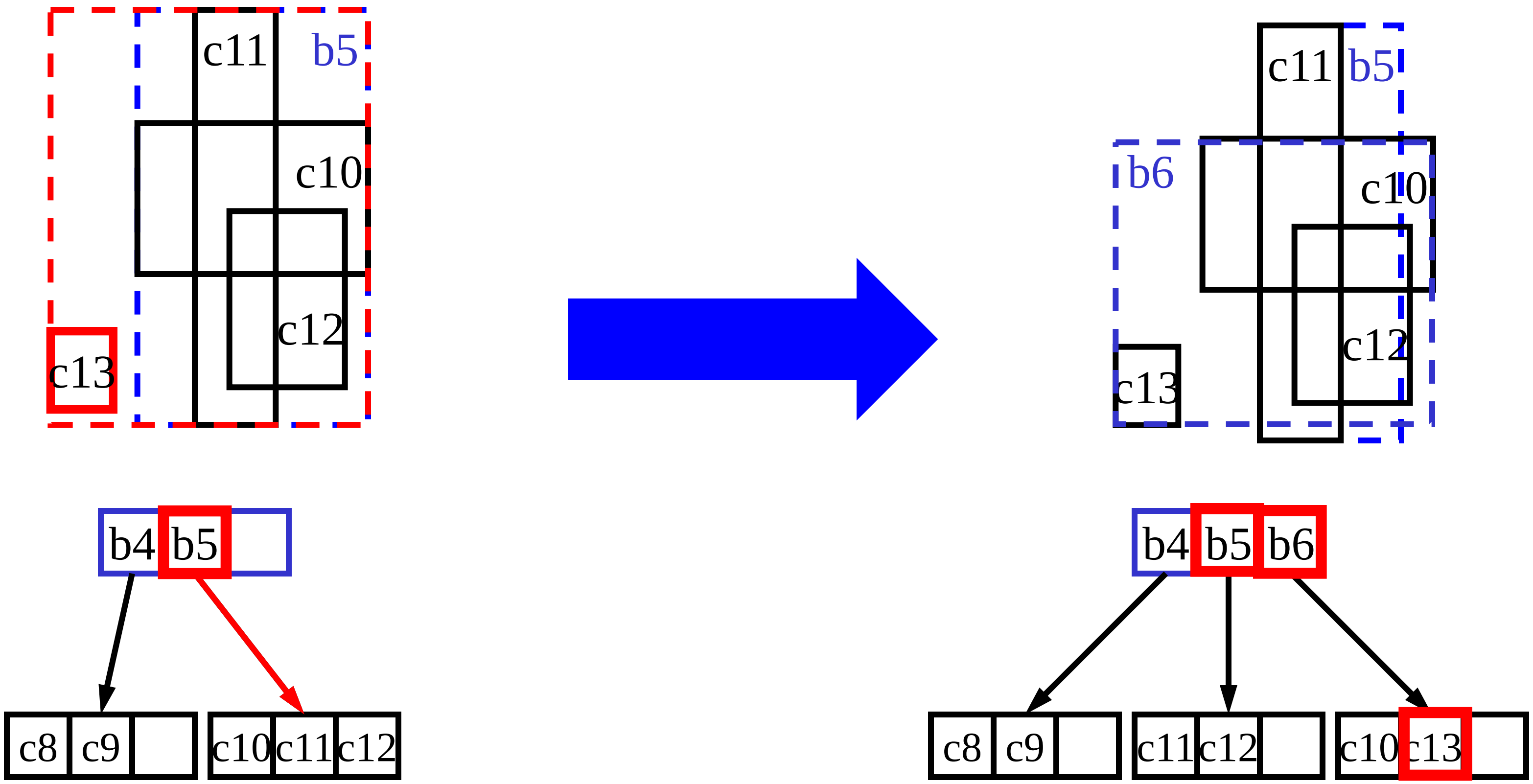




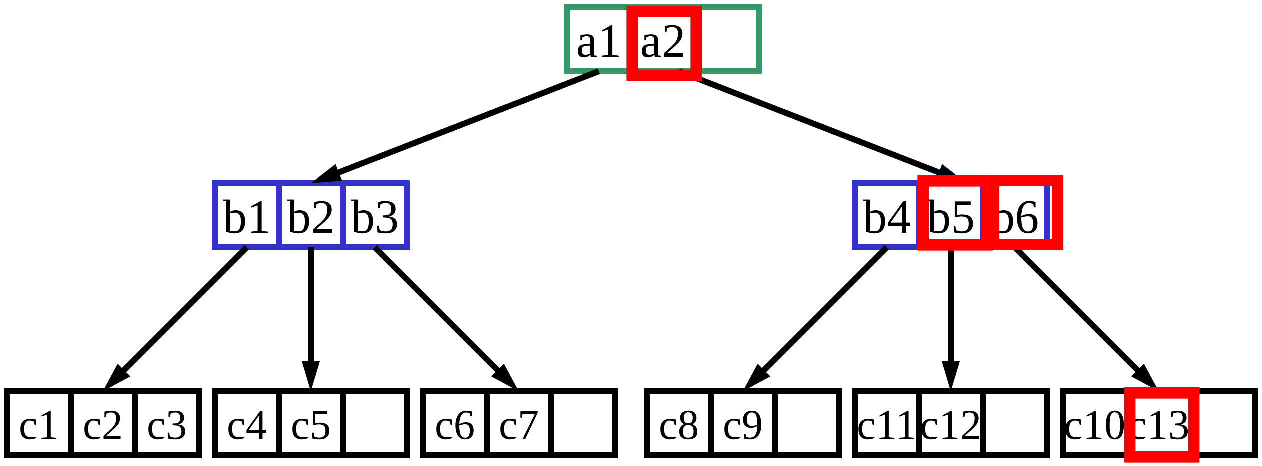
- Search regions whose MBB must be expanded the least
- Repeat on each level
- Here: Leaf overflow, split
  - Note: Choosing b4 would avoid split – but how can we know?

Example: Insertion, Split Phase

Several splits are possible



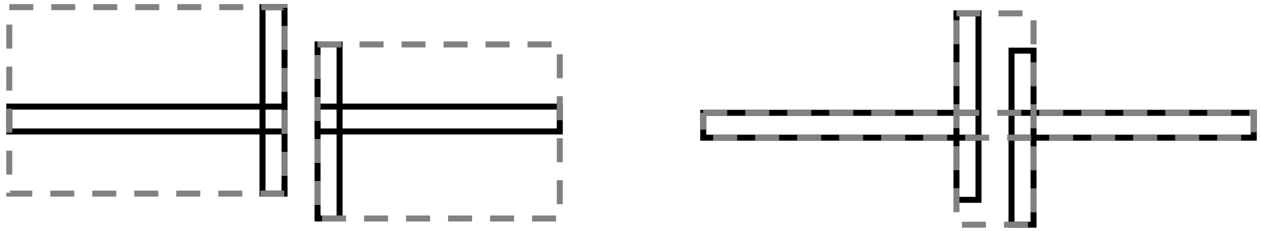
Example: Insertion, Adaptation Phase



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

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

- Complexity
  - 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
- Original Strategies (Minimizing Overlap)
  - Linear: Pick two objects farthest away. 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

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 leafs
  – 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
- Example: Nearest neighbor image search

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 methods
- Some obvious problems
  - Points need more coordinates – 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

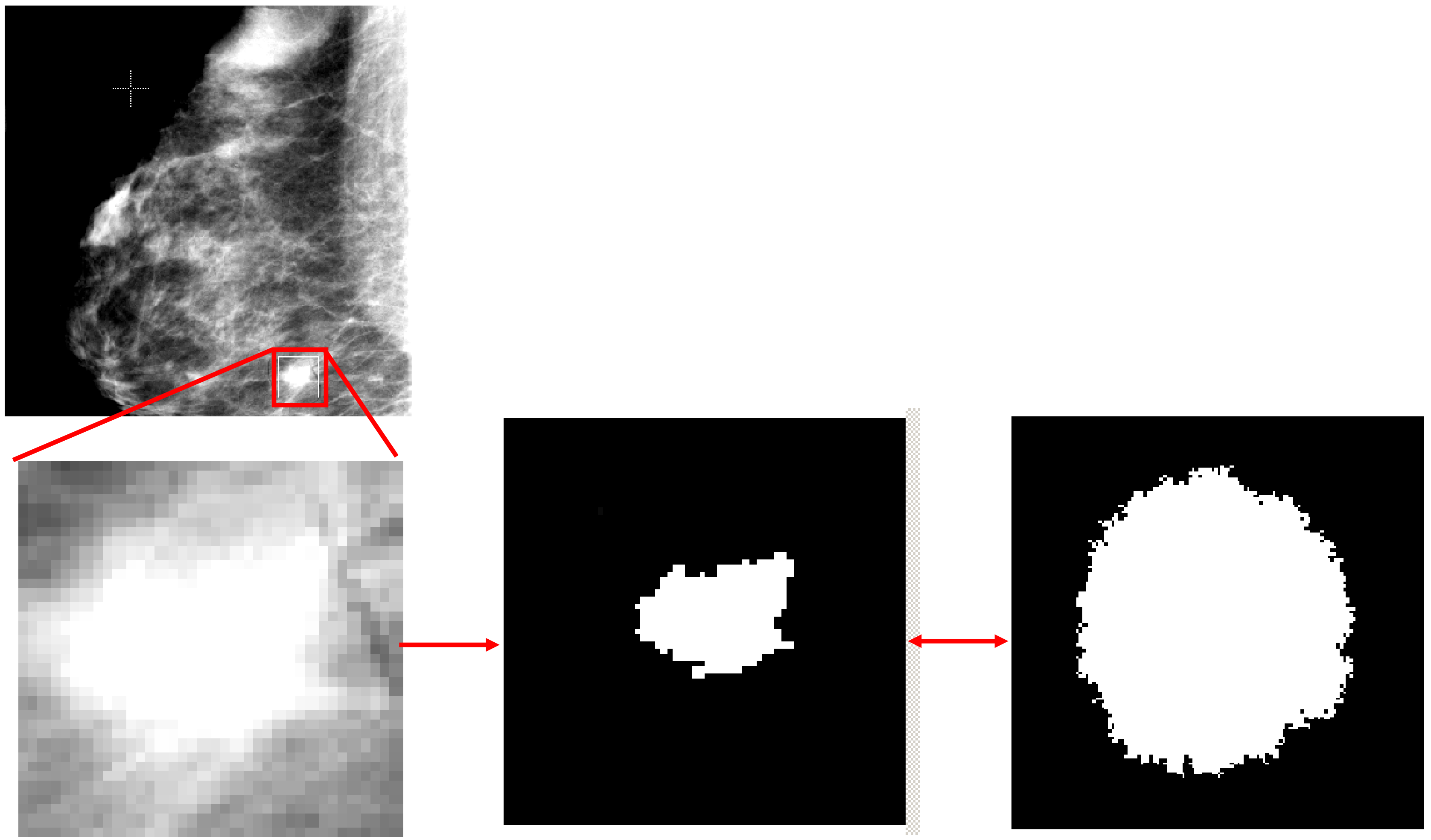
Curse of Dimensionality – Consider a growing d

- Some less obvious mathematical facts
  - Weber, R., Scheck, H. and Blott, S. (1998). "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces". VLDB
- 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

Content of this Lecture

- Introduction
- Partitioned Hashing
- Grid Files
- kdb Trees
- R Trees
- Conclusions
- 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.
  - Seidl, Kriegel (1998): Optimal Multi-Step k-Nearest Neighbor Search, SIGMOD.

Similar Objects in Images



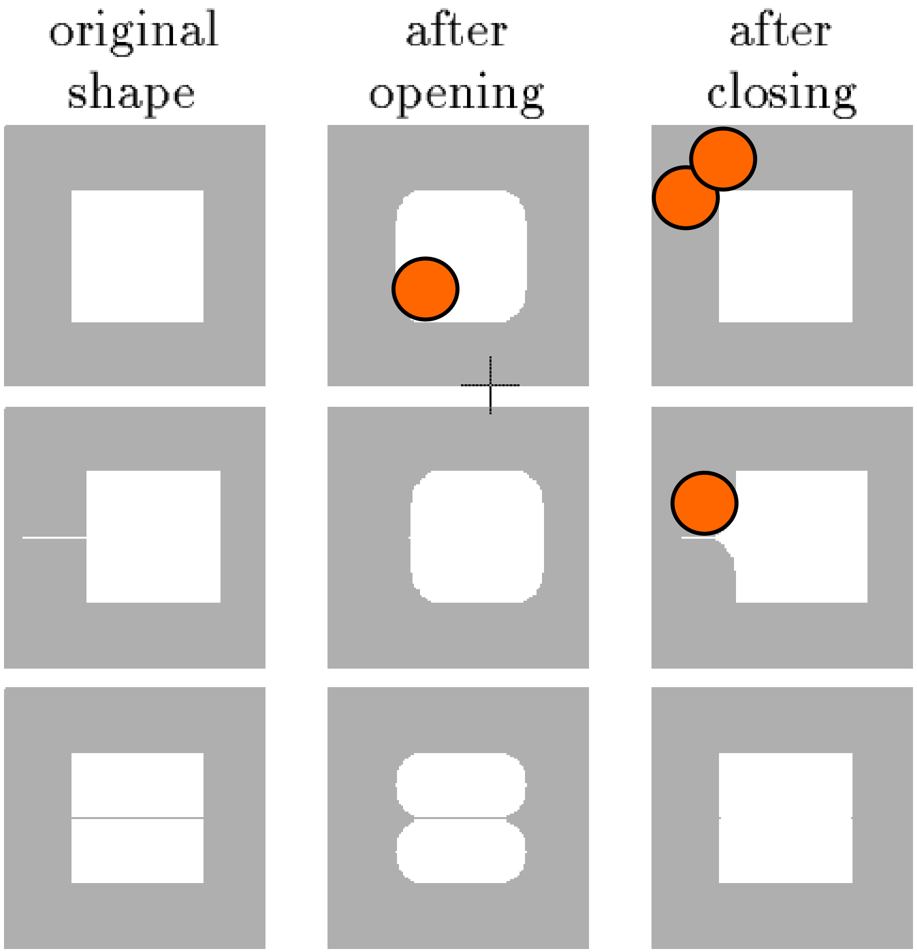
2D Object Similarity Search

- Similarity search: Fast algorithm to find all similar / the most similar objects in a database of objects
- Brute force: Compare against all objects
- Consider a visual-based distance function
  - Shape, size, rotation, borders, ...
  - Non trivial to express this as a vector distance function
  - How could we use a MDIS?
- 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



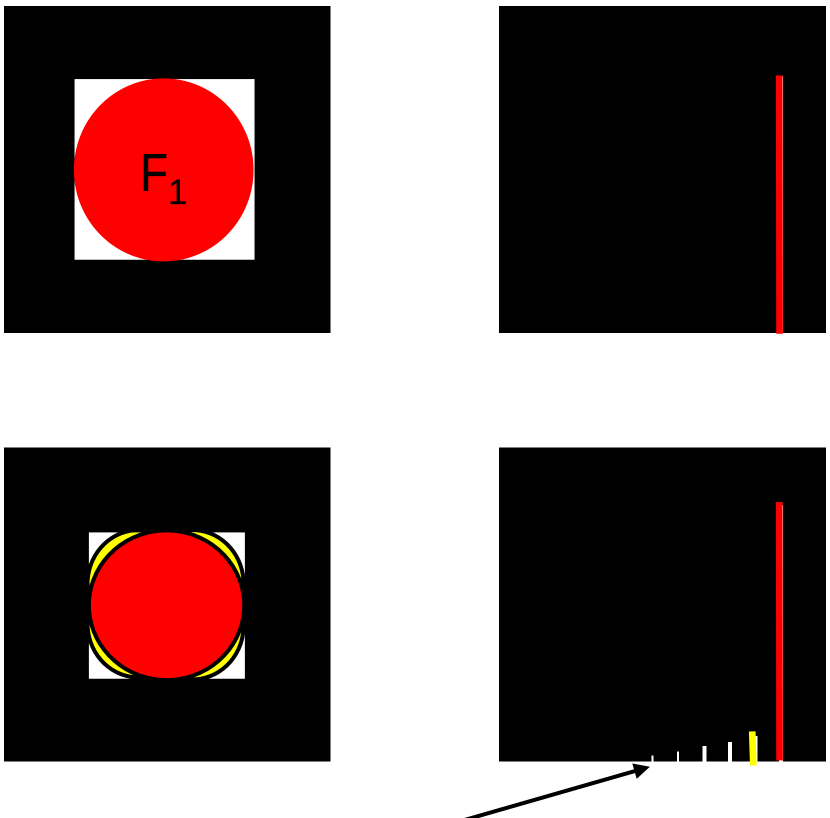
Distance Function

- Overlay objects o_1 and o_2
  - Align centers of mass
  - Rotate until maximal overlap
- Assume we use n different 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 $dist_i(o_1,o_2)=max((O_{1i} \cap O_{2i})/(O_{1i} \cup O_{2i}), (C_{1i} \cap C_{2i})/(C_{1i} \cup C_{2i})))$
- Define $dist(o_1,o_2) = max(dist_1(o_1,o_2), \ldots. dist_n(o_1,o_2))$

Scalability

- Very precise method (compared to human intuition)
  - Adaptable by varying n / thickness of brushes
- Highly complex -> very slow
  - Multiple computations of spatial overlaps between irregular shapes
  - Cannot be used to search against thousands of objects
- Idea
  - Find a distance function d' such that $d'(o_1,o_2) \sim dist(o_1, o_2)$ but $d'(o_1,o_2) \leq dist(o_1, o_2)$
    - d' should approximate dist as good as possible but never overshoot
  - If we have a max distance t: If $d'(o_1, o_2) > t$, then $dist(o_1, o_2) > t$
  - Idea: Use d' for pruning
    - Only helps if $d'(o_1, o_2)$ is (a) fast and (b) approximates dist well

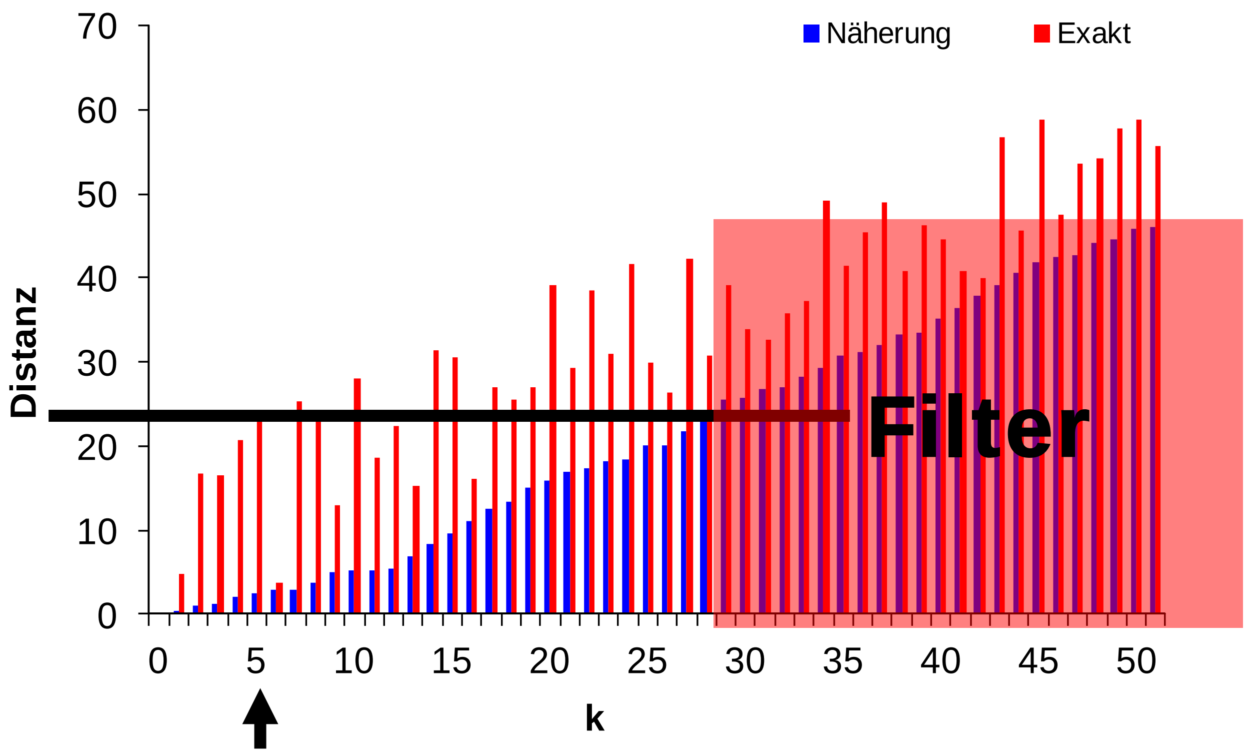
Spectrum Function



Spectrum = [...; 0.5; 0.8; 1.5; 5]^T

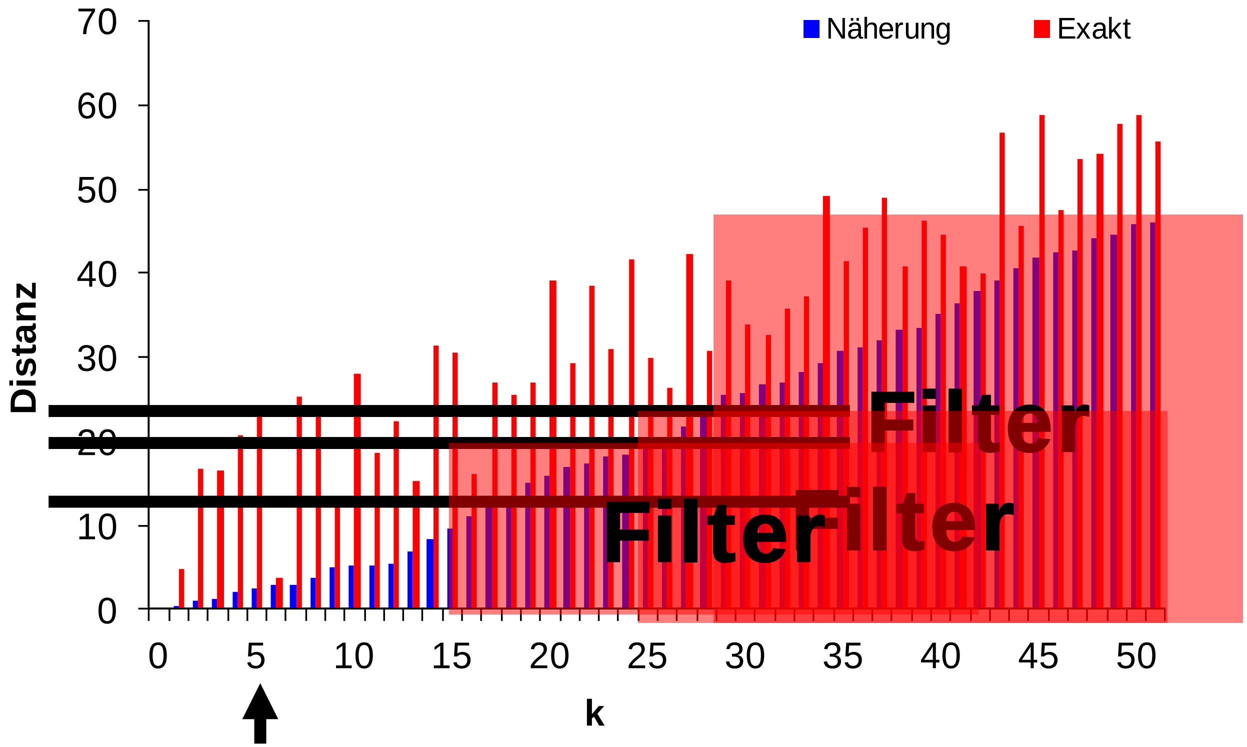
- Consider values O_{11}, O_{21}, ... O_{n1} (and C_{11}, ...)
- Compute spectrum: Vector with differences O_{11}-O_{21}, O_{21}-O_{31}, ...
- Euclidian distance between two spectra is a lower bound for true distance function dist

Intuition: 5NN Search



- Find the 5-furthest according to approximate distance d‘
- Compute maximum m of real distances
- Filter all objects with d’>m

Optimal: Iterative Refinement

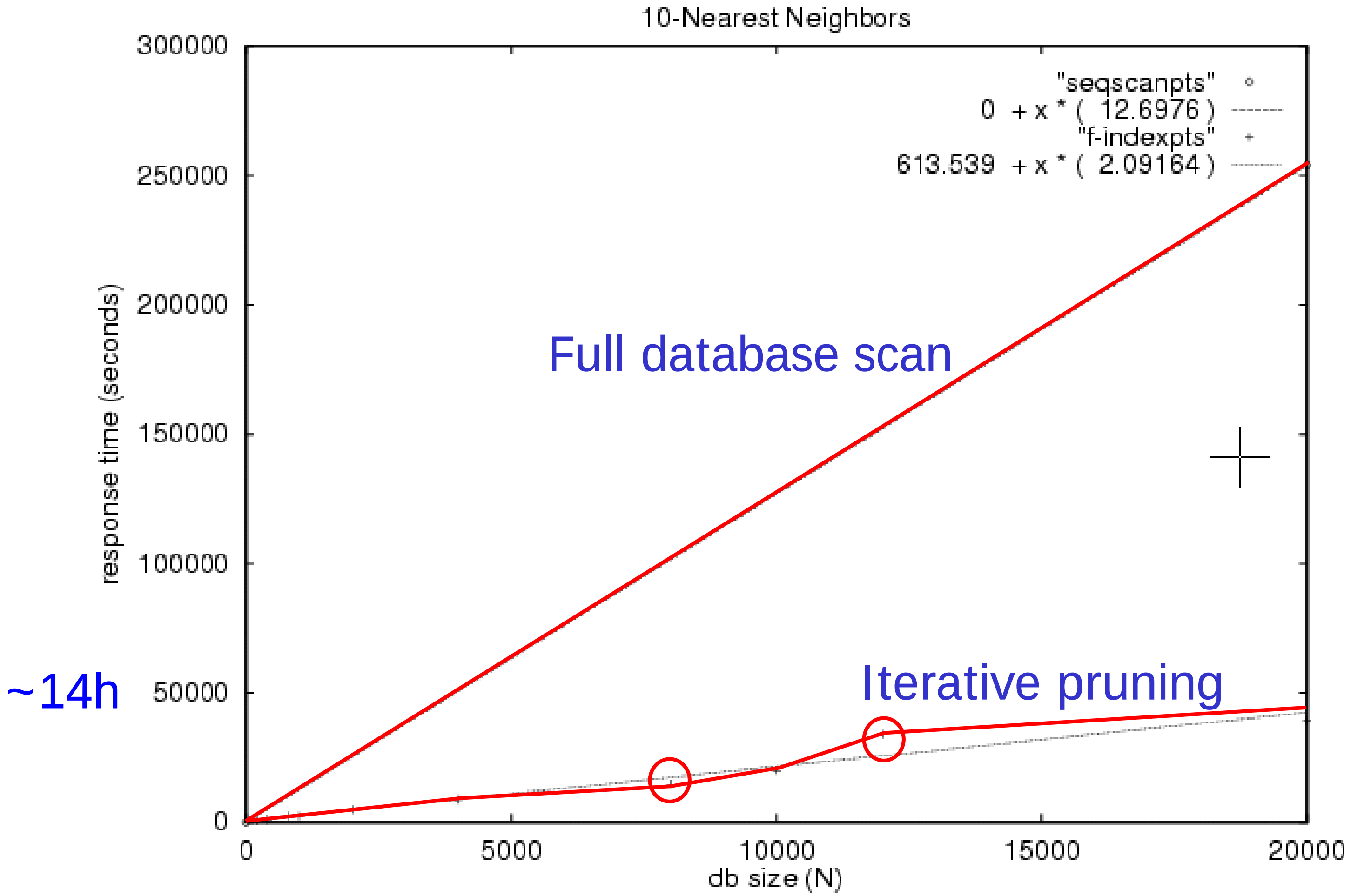


- Consider filtered objects in order of d'
- Whenever m gets smaller, prune again

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 fast approx search
  - Compute real distance and use as new t
  - Iteratively prunes search space

Effect



10-Nearest Neighbors
"seqscanpts"
0 + x * (12.6976)
"f-indexpts"
613.539 + x * (2.09164)
300000
250000
200000
150000
100000
50000
0
response time (seconds)
0
5000
10000
15000
20000
db size (N)
Full database scan
Iterative pruning
~14h