Datenbanksysteme II: Multidimensional Index Structures 2

Ulf Leser

Content of this Lecture

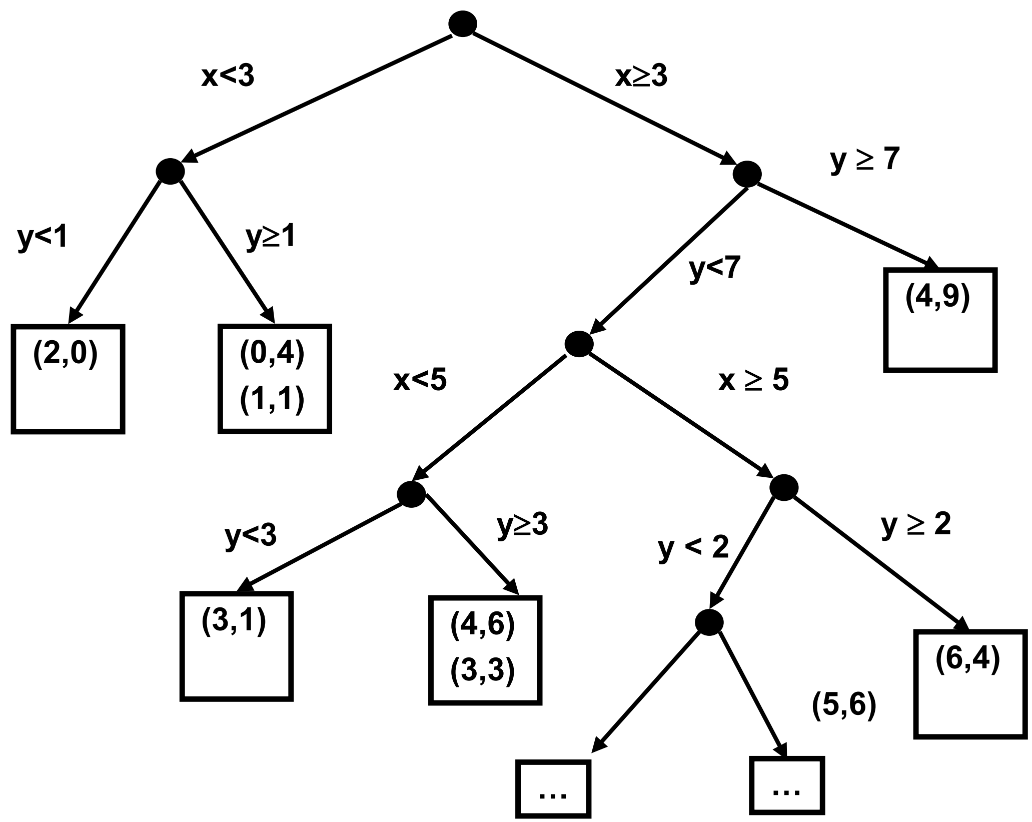
- Introduction
- Partitioned Hashing
- Grid Files
- kdb Trees
  - kd Tree
  - kdb Tree
- R Trees

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
  - 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

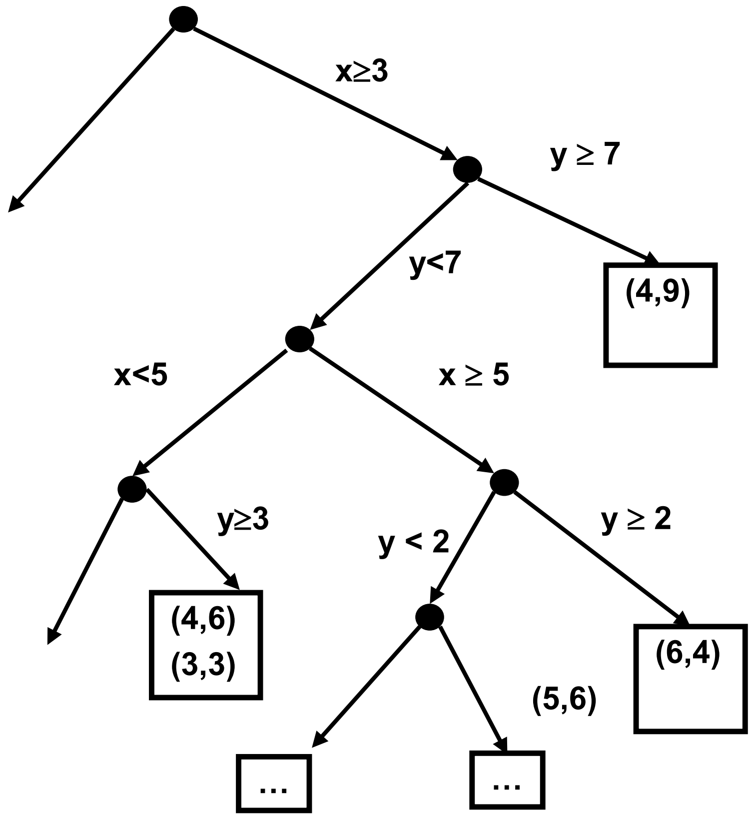
General Idea

- Binary, rooted tree
- Inner nodes define splits (dimension / value)
- Dimensions may be mixed in same level
- Leaves: Values + TIDs
- Each leaf (at depth m) represents a d-dimensional convex hypercube
  - With m≤2d border planes
- Not balanced
  - Bad WC search

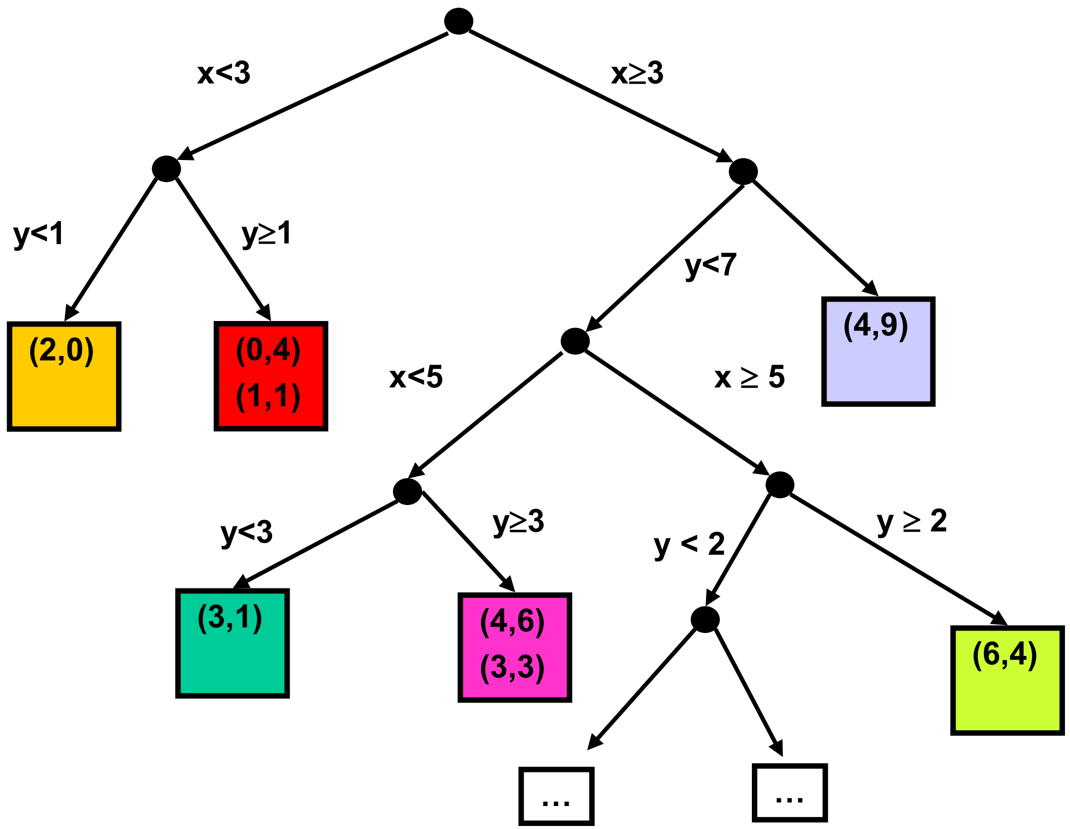


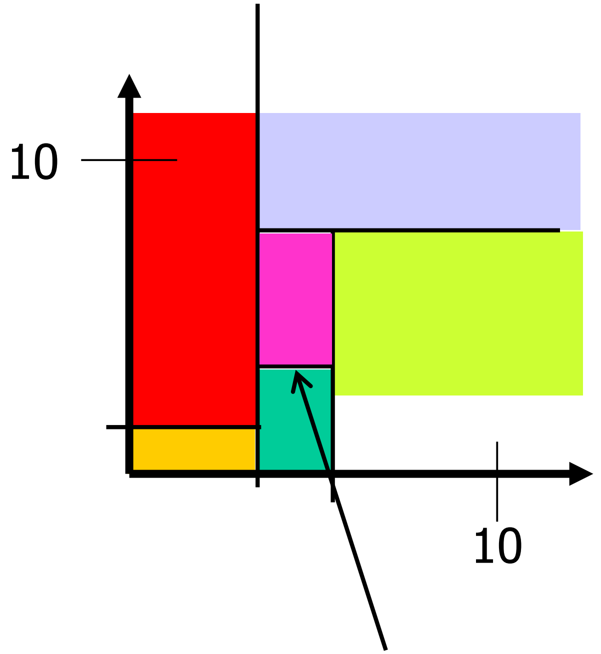
Main Memory or Secondary Storage?

- Keep everything in memory
  - Leaves are singular points
- Tree in mem and blocks on disk
  - Splits are delayed until block overflows
- Store everything on disk
  - kdb tree: Later
- On modern hardware
  - Random mem access in inner tree
  - Larger leaves create smaller trees
  - Parallel search? SIMD?
  - BB-Tree: Later



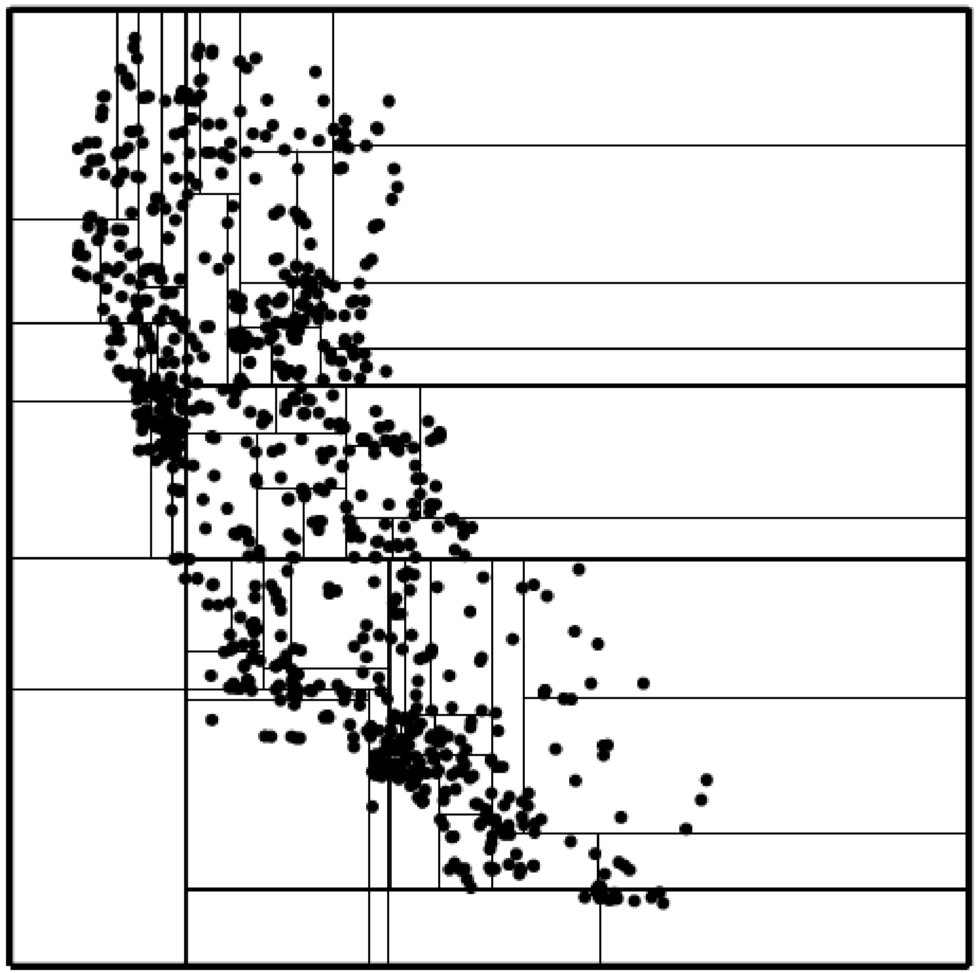
The Brick Wall





- 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
  - ?

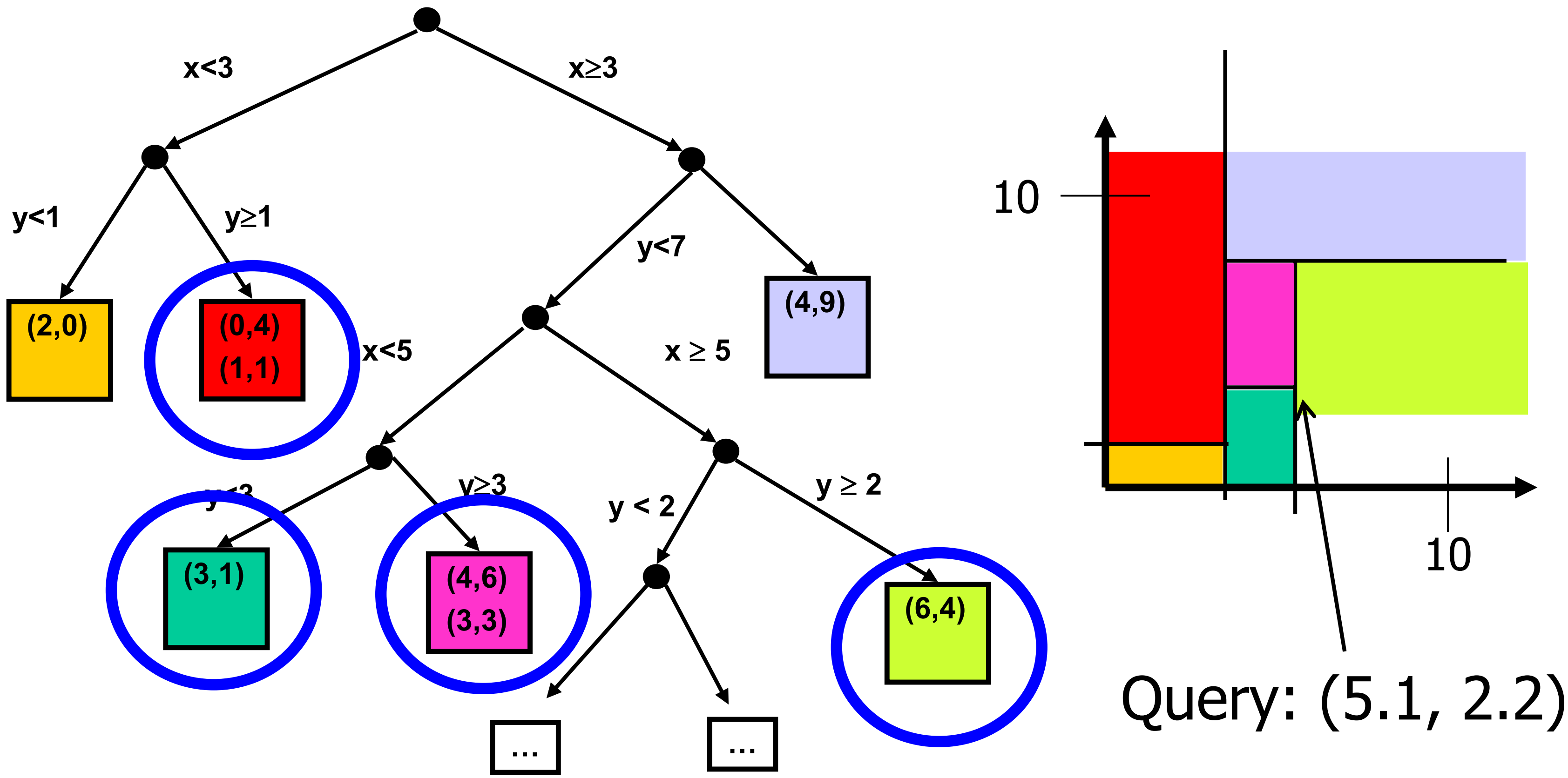
Search Operations

- 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 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 (nothing to exclude) searches 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



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
  - 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 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
  - Consider larger neighborhood: Grant parents, grant-grant-par …
- kd trees have no guaranteed balance (~ depth)
- There is no guaranteed fill degree

Static kd Trees

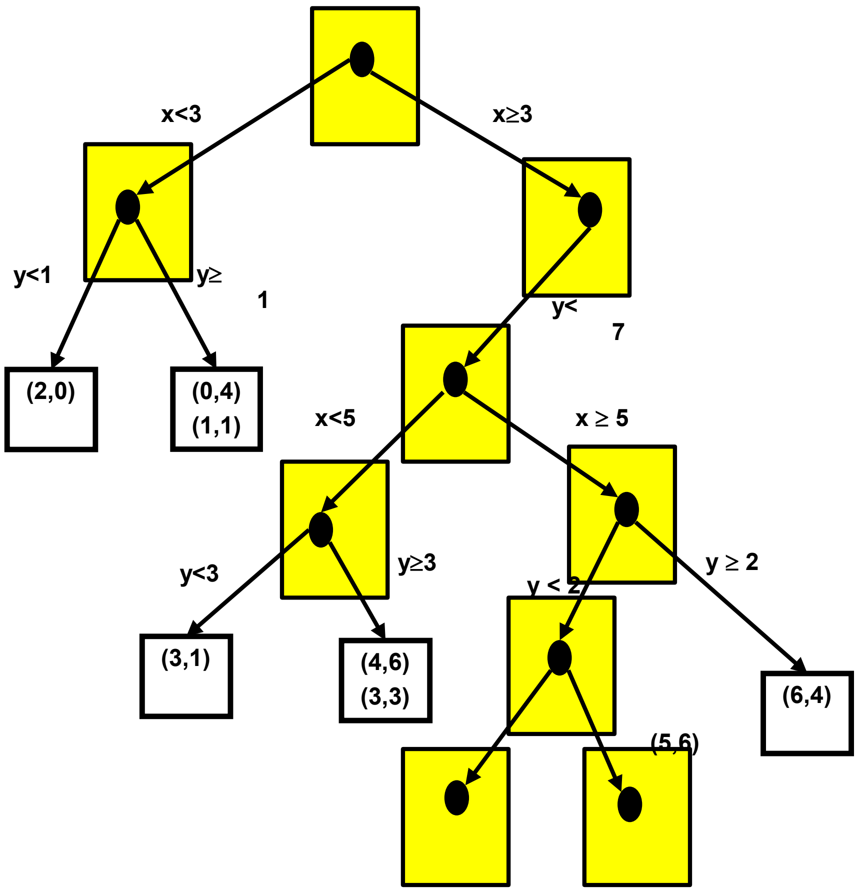
- Assume the set of points to be indexed is static and known
- We can build worst-case optimal kd Trees
  - Rotate through dimensions
    - Typically in order of variance – wide-spread dimensions first
  - Sort remaining points and choose median as split point
  - Guarantees tree depth of O(log(n)) for 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

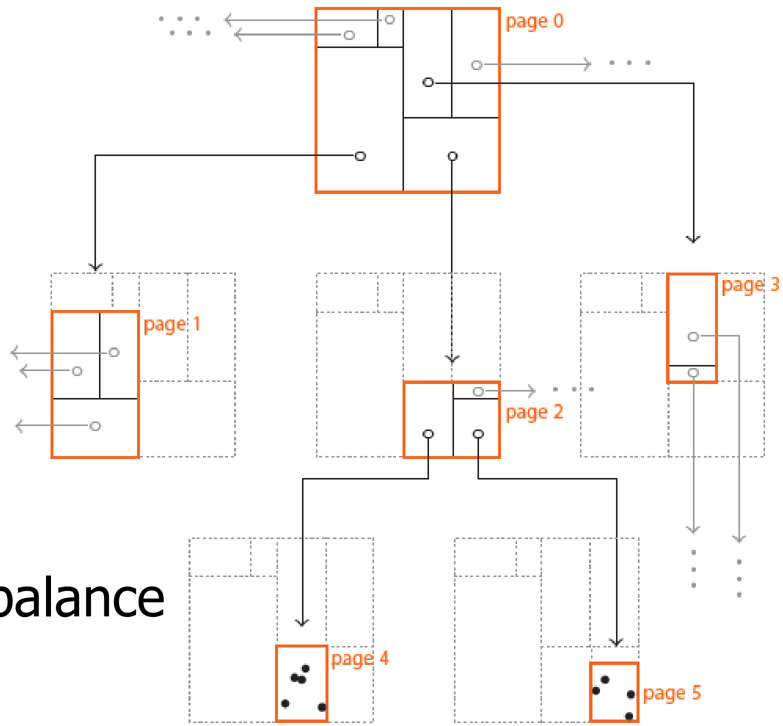
kd Trees on Secondary Storage – Naive Solution

- Each leaf is one block
- Store each inner node in one block
  - Inner blocks are essentially empty
  - Since tree is not balanced, worst case requires O(n) IO

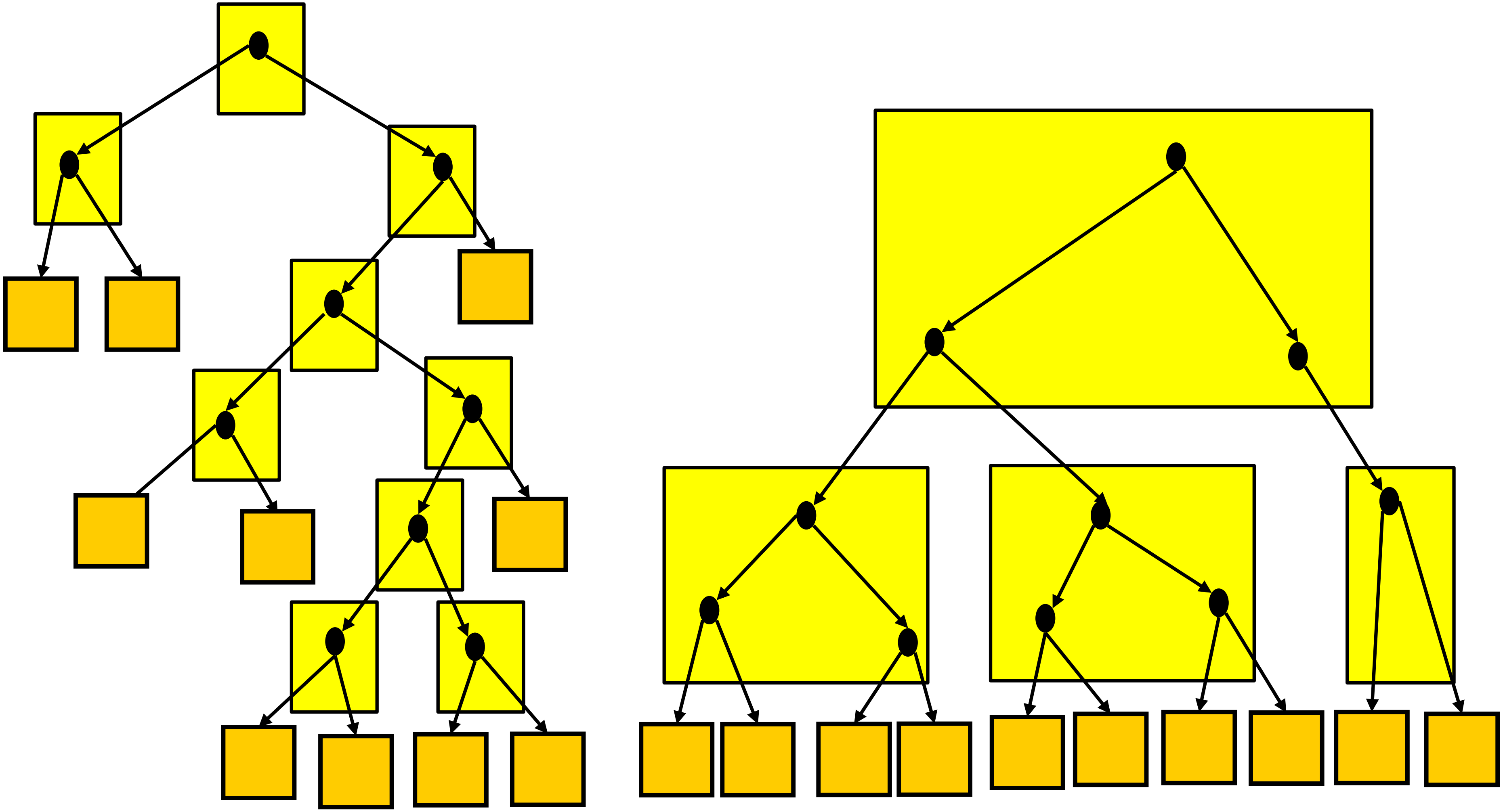


kdb trees

- Map many inner nodes to a single blocks
  - Robinson: The kdb-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
  - Block tree has balanced height
  - No guaranteed fill degree
- Operations
  - Searching: As with kd trees, but has balanced depth
  - Insertion/Deletion: Keep block tree balance



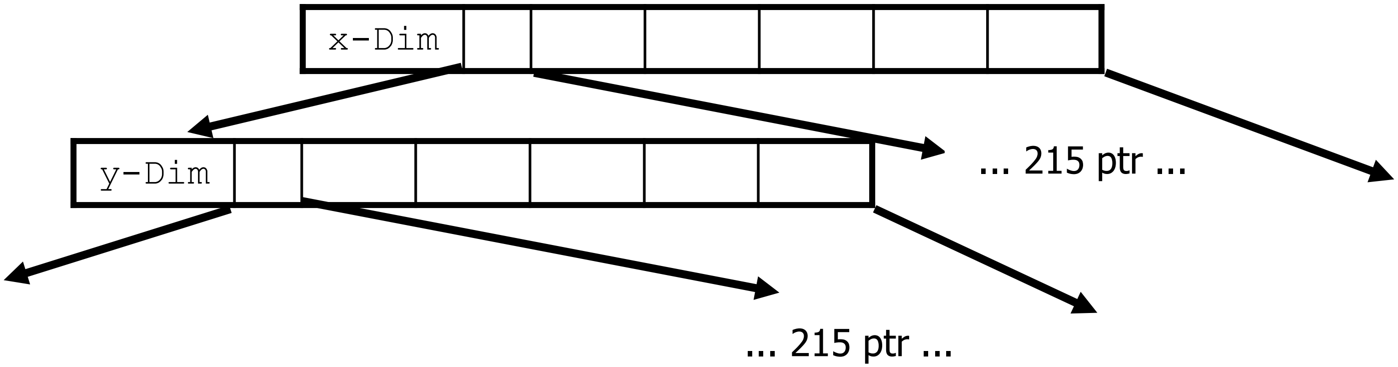
Sketch



Example – Composite Index

- d=3, n=1E9, block size 4096, |point|=9, |b-ptr|=10
  - App 450 points per leaf block ~ we need ~2.2M leaf blocks
  - Uniform distribution
- 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
- 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 ~ 10*215*0,05~100 IO
  - For each, we read 5% of data blocks = 1150 IO
  - Altogether: ~1250 IO
  - Optimal: Selectivity is 0.05^3 ~ 125K points ~ 270 IO

Visualization



Example: Partial Box Query

- Partial 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 1^{st} dimension
  - For each, we read 5% of 3rd level blocks = 2300 blocks
  - For each, we read 5% of data blocks = ~25K data blocks
  - Altogether: 27.000 IO
  - Optimal* 1E9*0,05*0,05/455 ~ 5.500 blocks

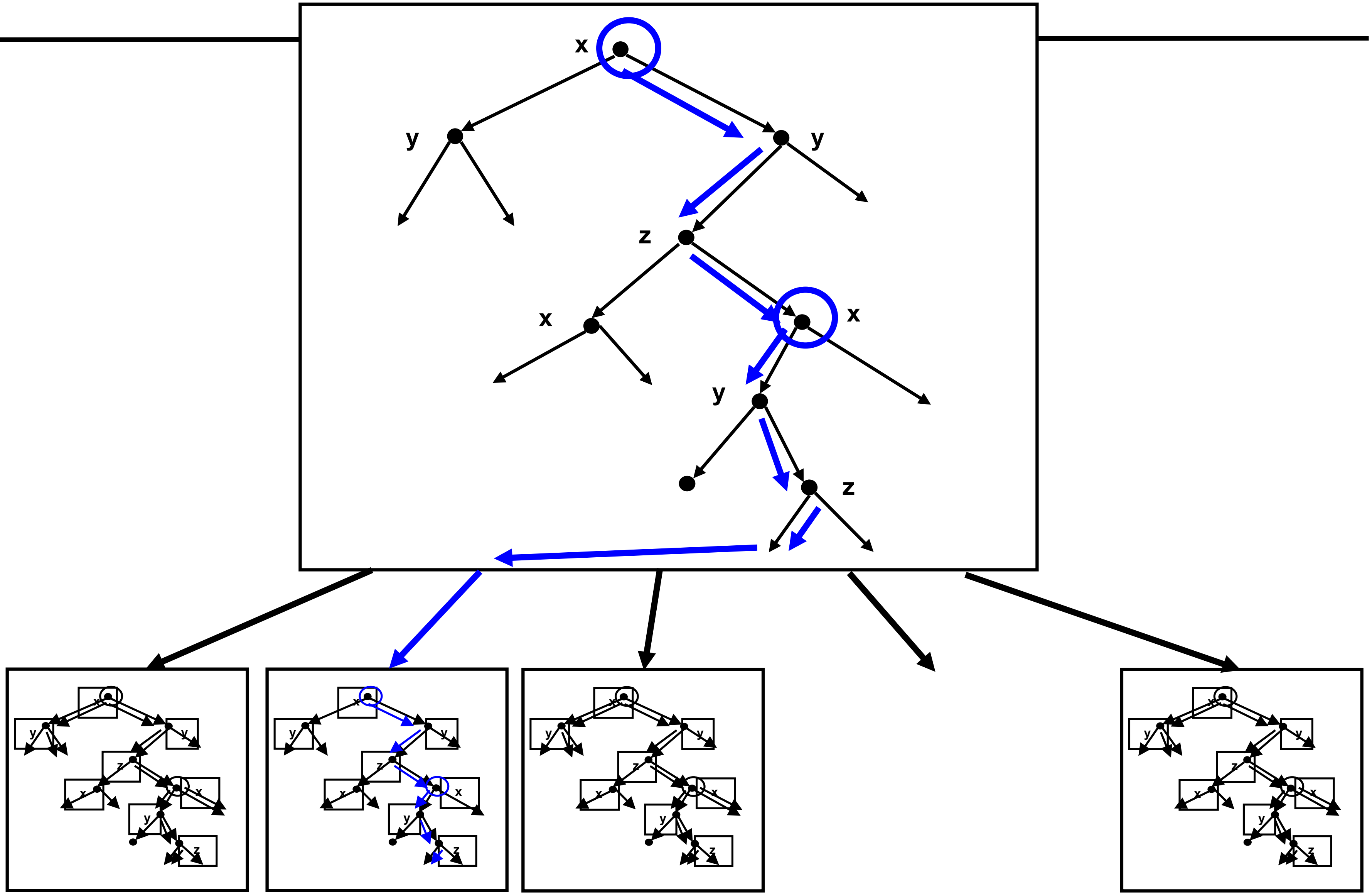
With Balanced kdb Tree

- Balanced kdb tree will have ~23 levels
  - 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
    - Last (7th) level has 64 nodes
  - There are 64 2nd level blocks holding levels 8-14 of kd-tree
    - Together, 64*64 = 4096 nodes at 14th level
  - There are ~4000 3rd level blocks holding levels 15-21 of kd-tree
  - There are ~260K 4th level blocks holding level 22-23
    - Together, app. 1M ~ 2^{21} leaves

Space Covered

- 1st block splits space in 64 regions
- 2nd level block split space in ~4K regions, each region covering 0,00025% of all points
- Query selectivity is $(0,05)^3$ = 0,000125% of points
  - Always assuming uniform distribution
- Thus, we very likely find all results in one region of first two levels and require increasingly more outgoing nodes in 3rd and 4th level

Intuition



Example - Box Query with kdb Tree

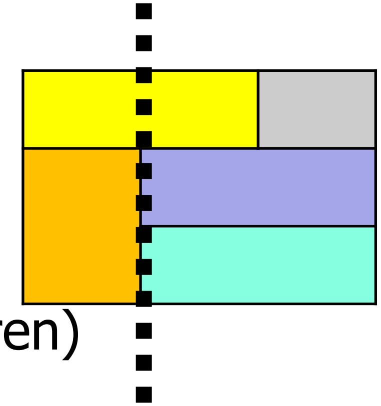
- Box query on thee dimensions, asking for a 5% range in each dimension
  - In first block (7 levels), we have 2 splits for 2 and 3 for 1 dim
  - In a box query, we know where to go in all splits
  - We need to check only 1 second-level block
  - In level 3, some splits are within query range
    - Let's assume two: $1*2^2$ = 4 blocks (of 64)
  - In level 4, more splits are within query range
    - Let's assume three: $4*2^3$ = 64 blocks (of 4096)
  - Level 4 blocks have 4 outgoing pointer: 4*64 = 256
  - Altogether: 1+4+64+256 ~ 320 IO
    - Compare to 1250 of composite index
    - Compare to 270 in optimal case

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 for 2 and 3 for 1 dim
  - Assume bad luck – no range for the 4-split dimension
  - We need to check 2^3=8 second-level blocks (of 64)
  - In level 3, more splits are within query range
    - Let's assume four: $8*2^4$ = 128 blocks (of 4.096)
  - In level 4, more splits are within query range
    - Let's assume four again: $128*2^4$ ~ 2.000 blocks
  - Level 4 blocks have 4 outgoing pointer: 4*2.000 ~8.000
  - Altogether: 1+8+2.000+8.000 ~ 10.000 IO
    - Compare to 26.000 for composite index
    - Compare to ~5.500 for optimal

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 block-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
  - Propagate new pointers to parent (and their children)
  - Might lead to reorganization of entire tree



Conclusion

- Pro kdb trees
  - Conceptually nice, close to B-tree idea
  - Balanced tree depth – good WC performance for searching
  - May achieve optimal search performance
- Contra kdb
  - No guaranteed fill degree
    - Many insertions/deletions may lead to almost empty leaves
  - Keeping balance requires sporadic tree reorganizations
    - Runtime of single insert / delete operations become unpredictable
  - Difficult to implement
- Rarely used in practice

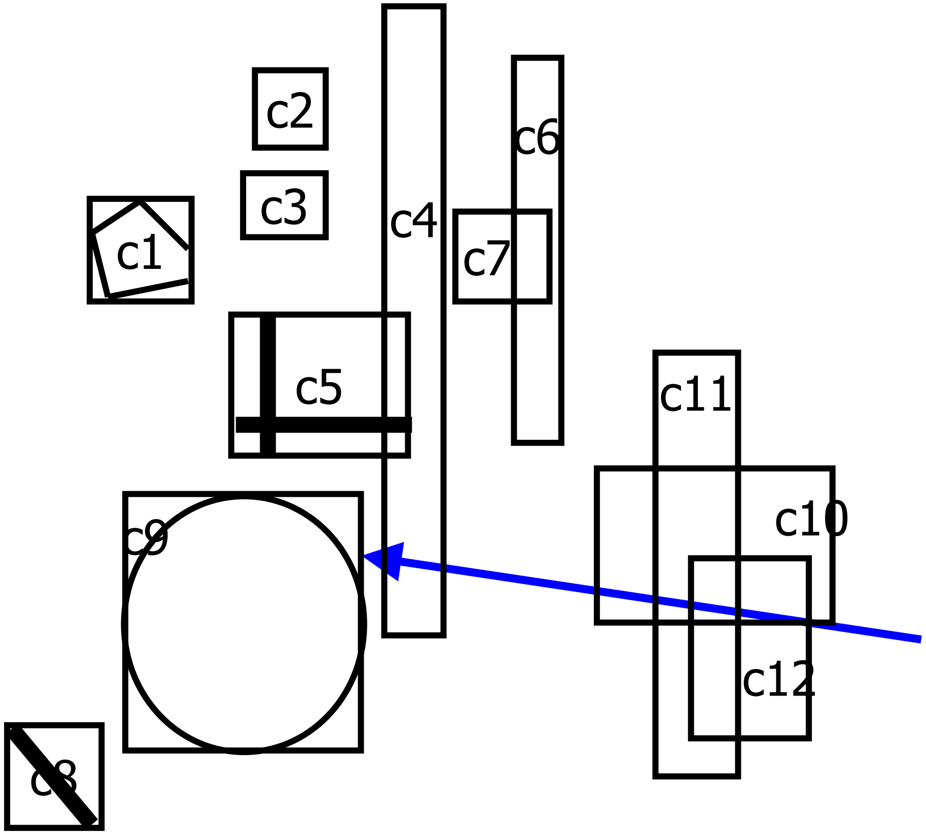
Content of this Lecture

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

R-Trees

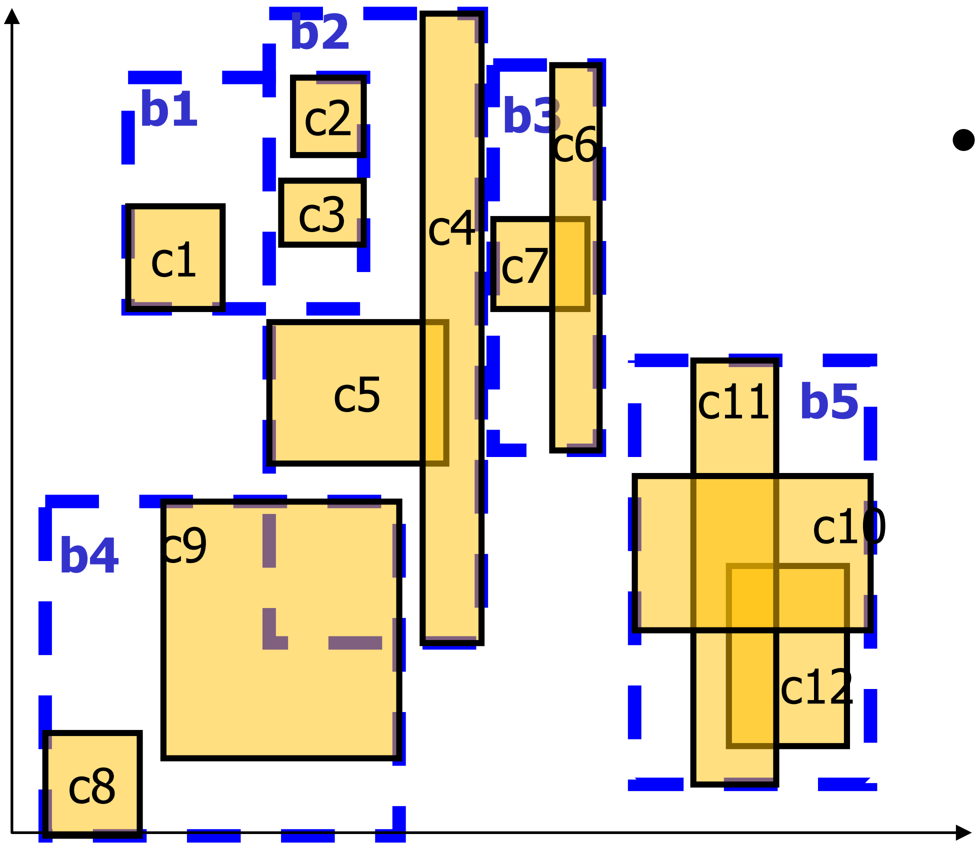
- Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD 1984.
- Can store geometric objects (with area) as well as points
  - Arbitrary geometric objects are represented by their minimal bounding box (MBB)
- Each object is stored in exactly one region on each level
- Since objects 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)

Example (from Donald Kossmann)

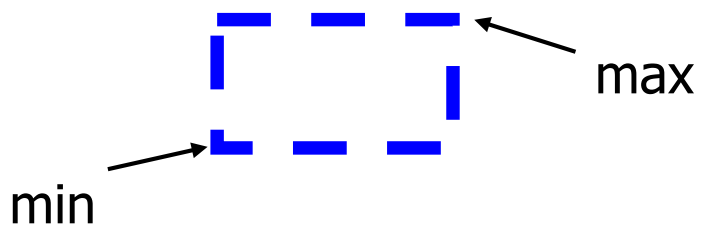


Compute MBBs for all non-rectangular objects

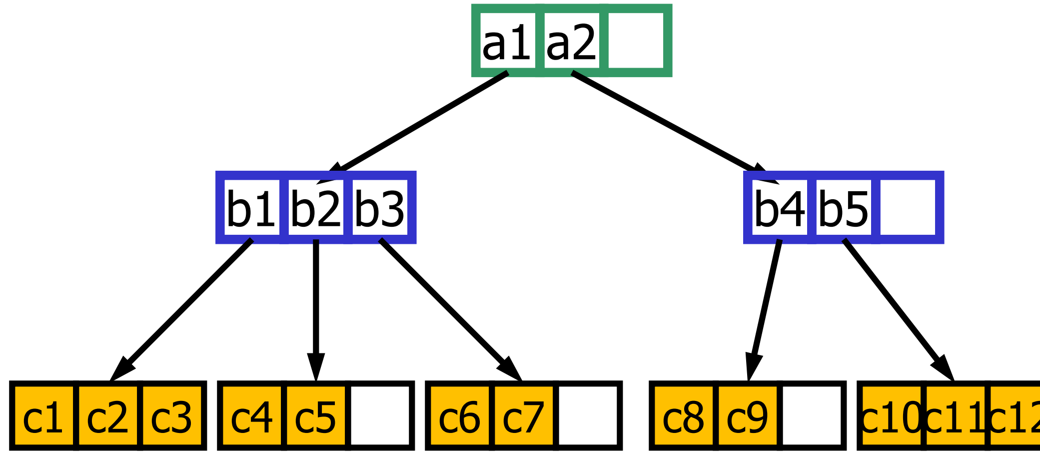
General Idea

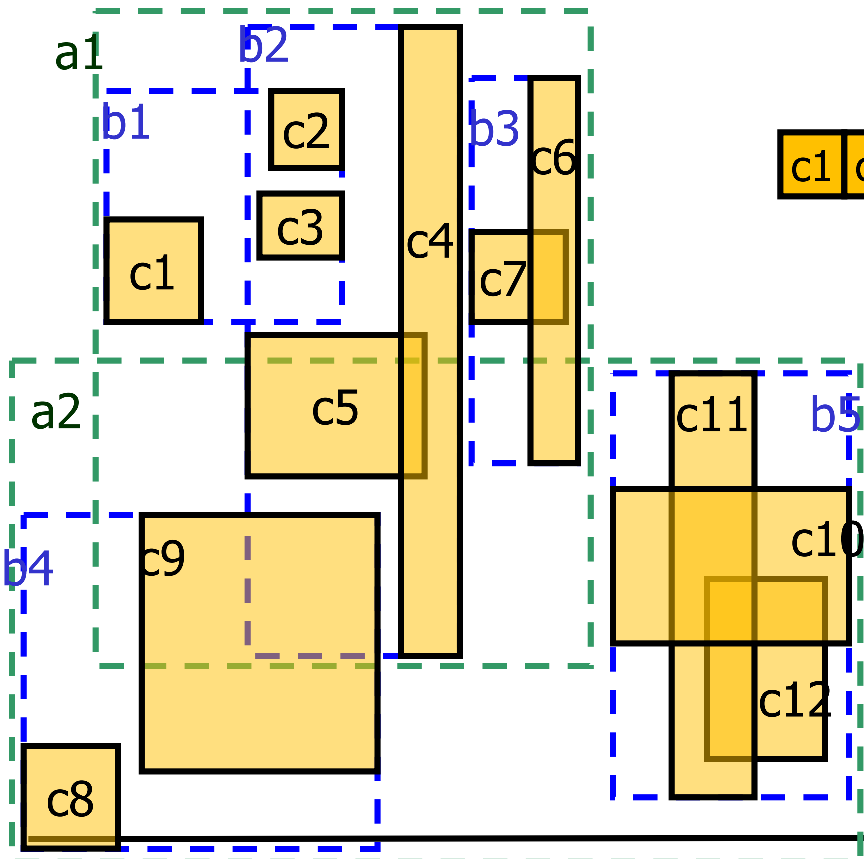


- We group clusters of spatial objects into minimal bounding box (MBB)
- Each MBB is represented by two corner points (in 2D, otherwise …)



General Idea

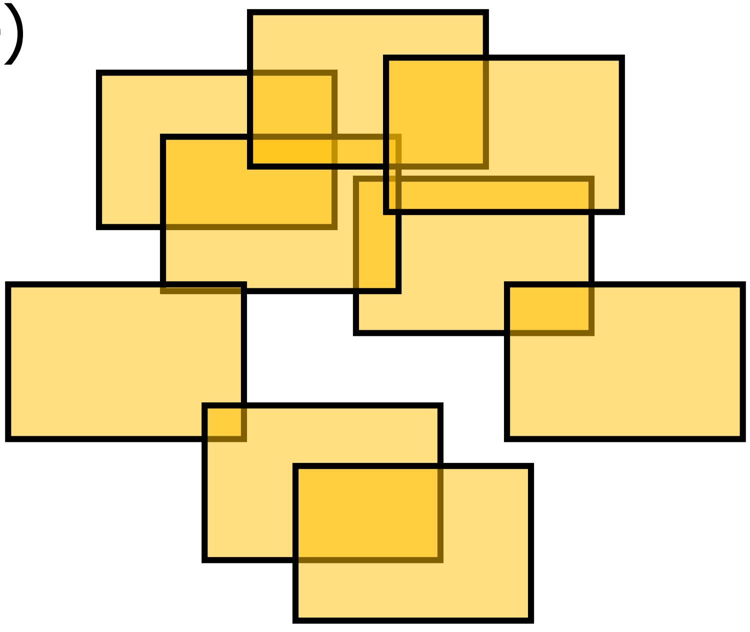




- Objects are stored only once in leaf nodes
- We group MBBs hierarchically into overlapping regions

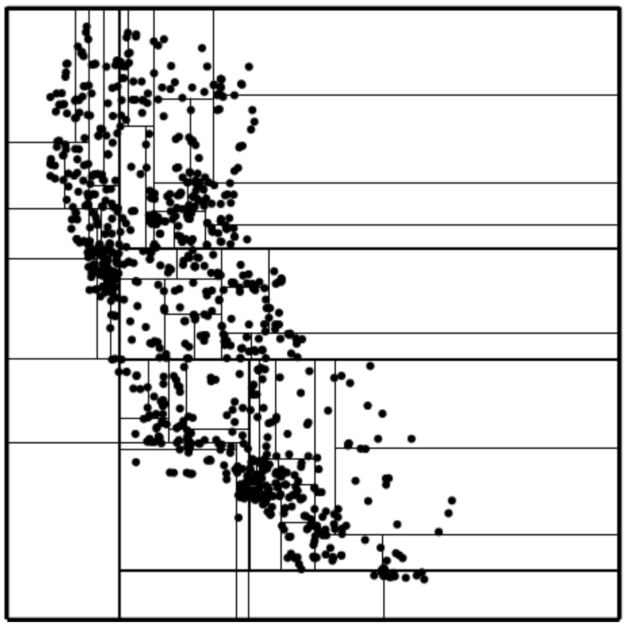
Motivation: Objects that are not points

- 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

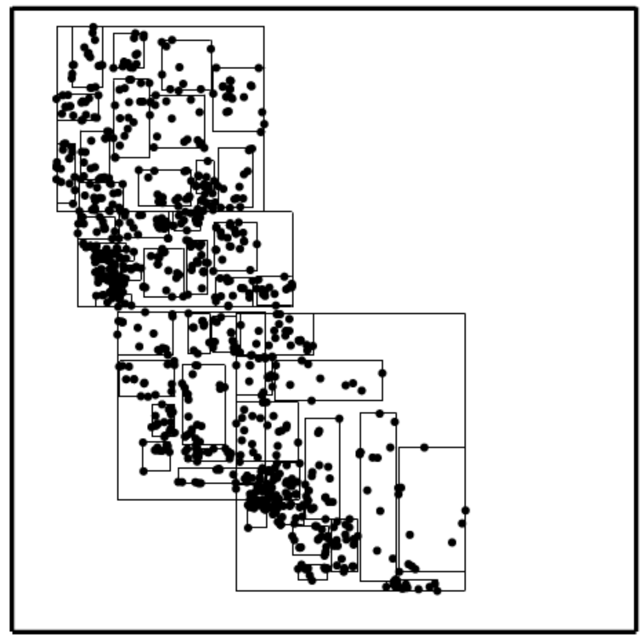


R Tree versus kd Tree

kd Tree



R Tree



Concepts

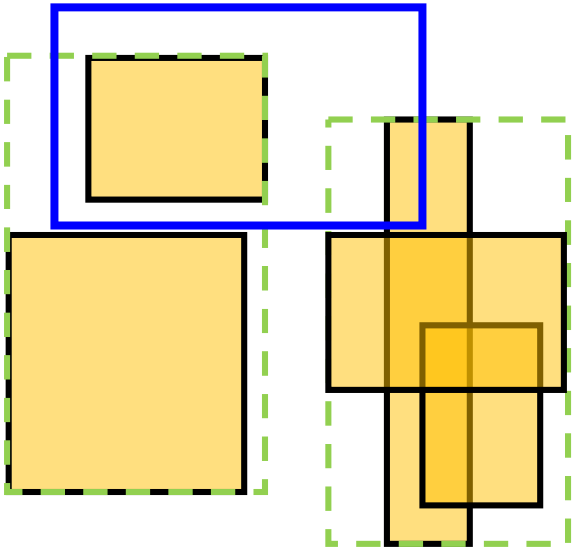
- Inner nodes consist of a set of d-dimensional regions
  - Every region is a (convex) hypercube – a MBB
- Regions are hierarchically organized
- Each region of an inner node points to a subtree or a leaf
- The region border is the MBB of all objects in this subtree
  - Inner node: MBB of all child regions
  - 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

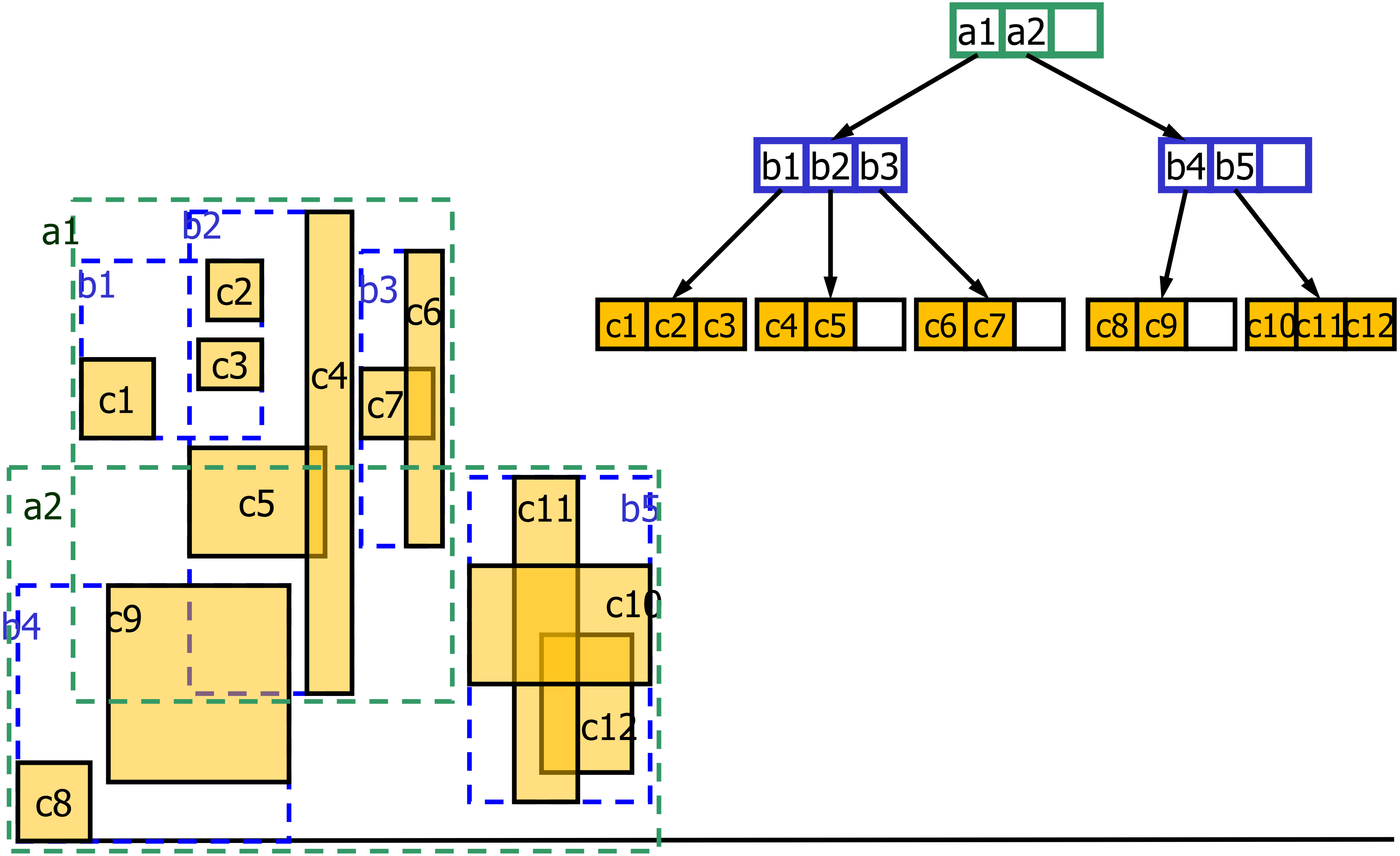
- Guaranteed fill degree: The number of regions of a node (except for the root) is between m and M
  - M : the maximum number of entries in a node
  - m: set to some fraction of M, e.g. M/2
- The root node has at least 2 entries
- Balanced: Leaf nodes are at the same level

Searching

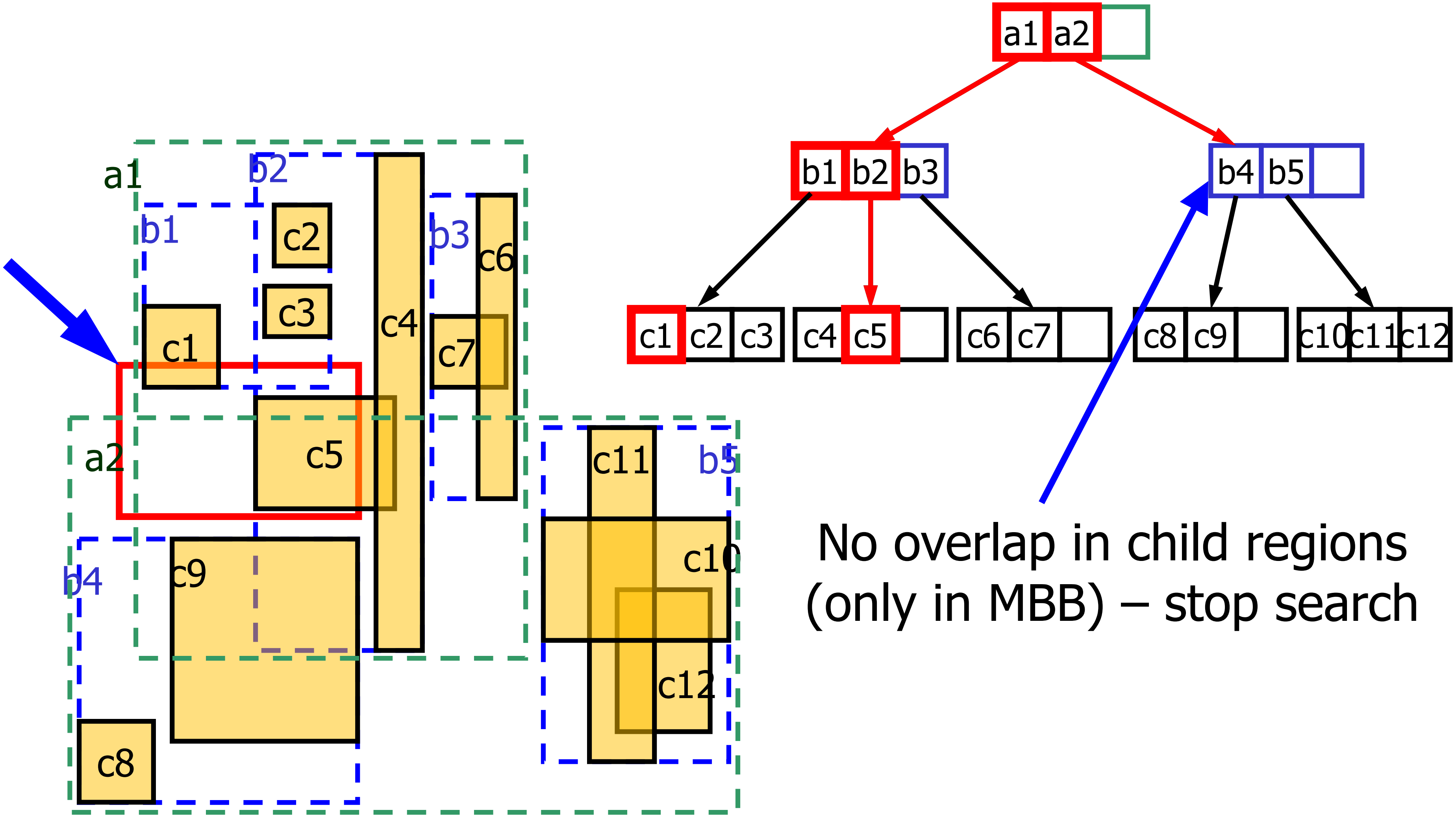
- Point query (for points as data objects)
  - At each inner node, find all regions containing the point
  - All those subtrees must be searched
- Box overlap query: Find all objects overlapping with a given query
  - In each node, intersect query with all regions
  - >1 region might have non-empty overlap
  - All those subtrees must be searched
- Box inclusion query: Find all objects within a given query object
  - Same as overlap query



One State



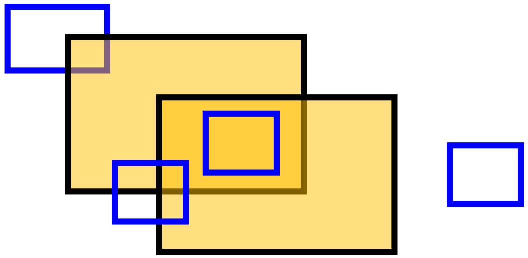
Example: Overlap Query



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
  - If at least one region with complete overlap
    - Choose one (smallest?) and descend
  - If none with complete, but at least one with partial overlap
    - Choose one (largest overlap?) and descend
  - If no overlapping region at all
    - Choose one (closest?) and descend

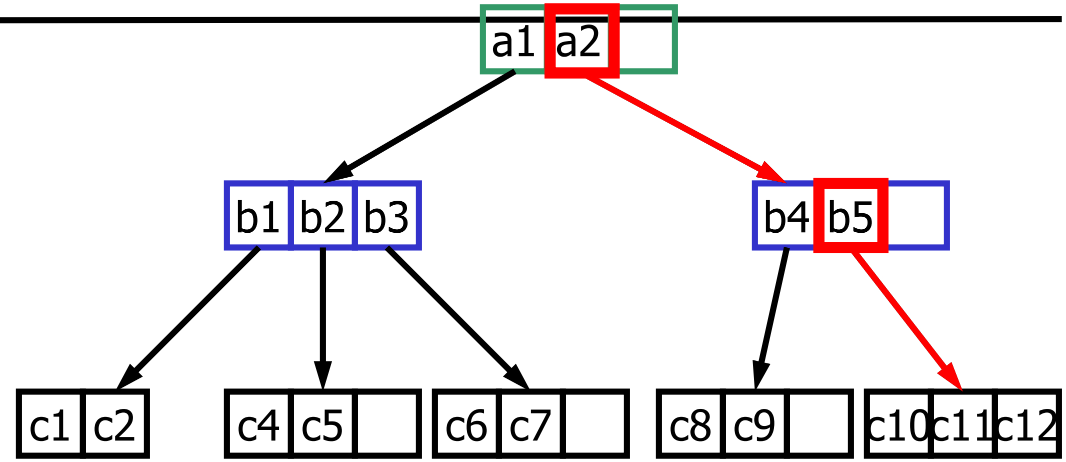


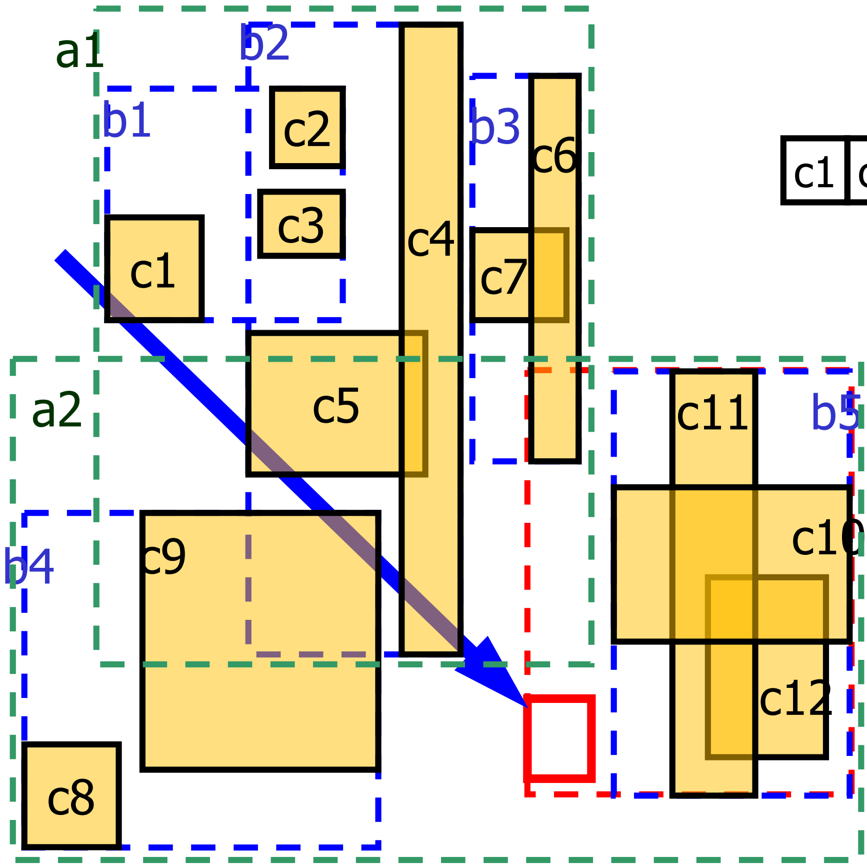
- Eventually, we reach a leaf
  - We insert object in only one leaf

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: Insertion, Search Phase

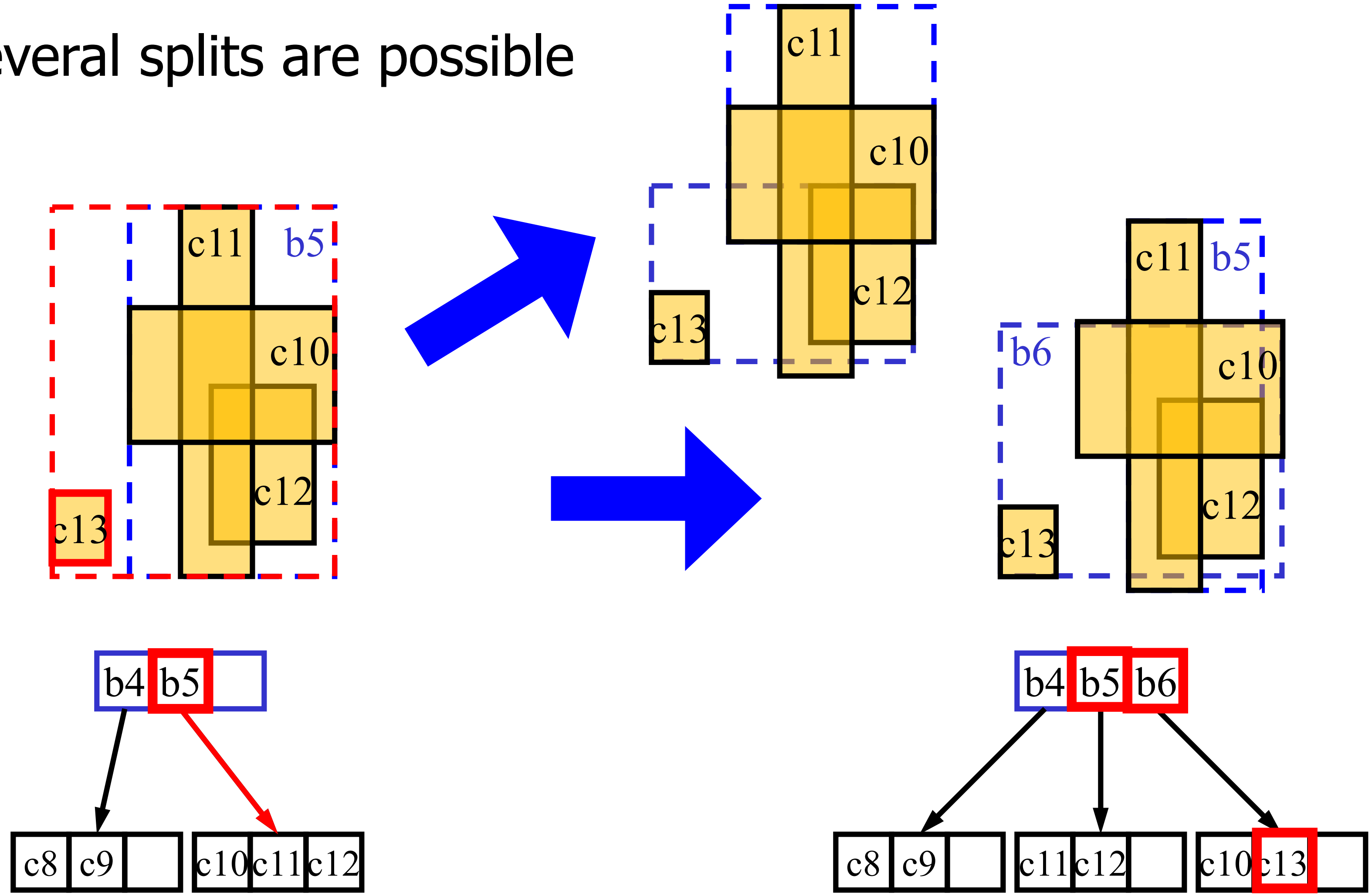




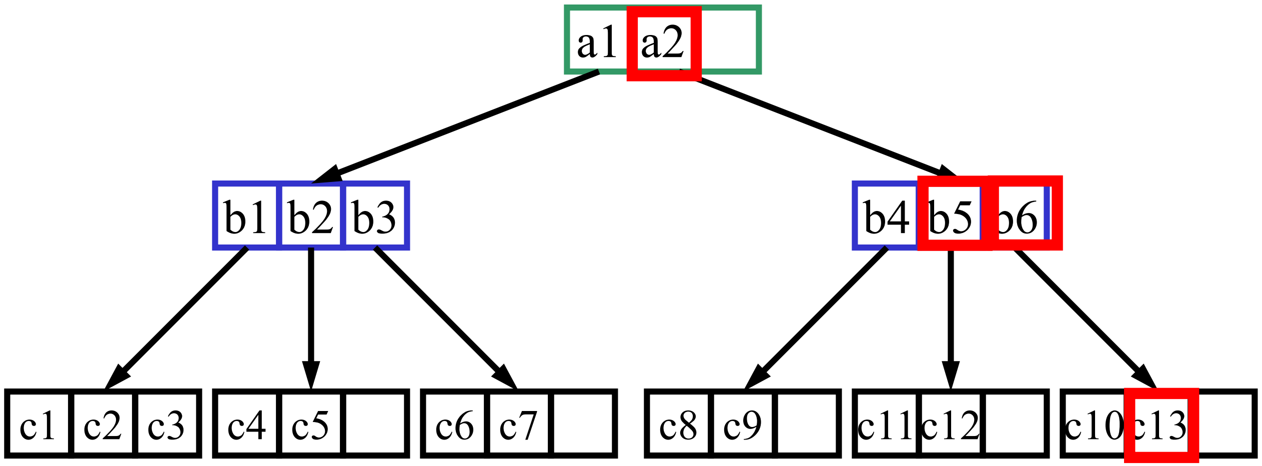
- Search regions whose MBB must be expanded the least
- Repeat on each level
- Here: Leaf overflow, split
  - Note: Choosing b4 would avoid split – but how can we know?

Example: Insertion, Split Phase

Several splits are possible



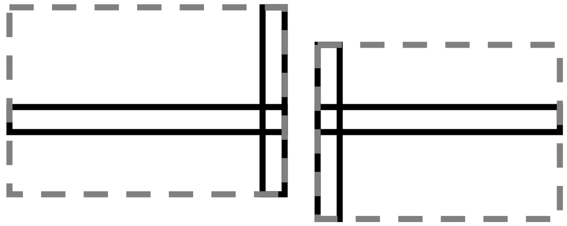
Example: Insertion, Adaptation Phase

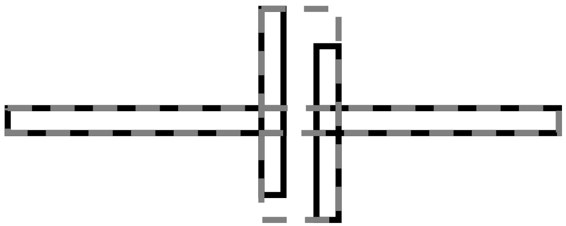


- MBBs of all parent nodes must be adapted
- Block split might induce node splits in higher levels of the tree (not here)

Where to Split

- Finding the best splitting strategy has seen ample research
- Wish 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
- Wish 2: Minimize covered space
  - 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





Deletions in the R Tree

- As usual: In case of underflow, 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 … expensive
- Trick: Delete by Reinsertion
  - Re-Insert every objects that remained in the underflown block
  - Insert strategies will be applied again
  - No particular delete strategy required – focus on good insertions
  - But costly: A single delete may incur many inserts
    - Depending on m

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
  - 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
  - Higher space consumption due to redundancy
  - Insertion may require down- and upward adaption
    - Like kdb Trees

R* Tree

- As Grid-files or kd-Trees, R Trees take decisions during insertions that determine the future of some regions
  - MBBs in chosen subtree change
  - During insertions, they usually grow
- If these decisions prove wrong, large overlapping MBBs emerge, making search slow
  - Too many branches need to be traversed
- R*: Revise your decisions from time to time
  - Chose regions and fraction of objects at random in regular intervals
  - Delete and reinsert
  - Leads to smaller MBBs and faster operations
    - Price: The unnecessary reinsertions

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