Data Warehousing und Data Mining

Multidimensionale Indexstrukturen

Ulf Leser

Wissensmanagement in der Bioinformatik

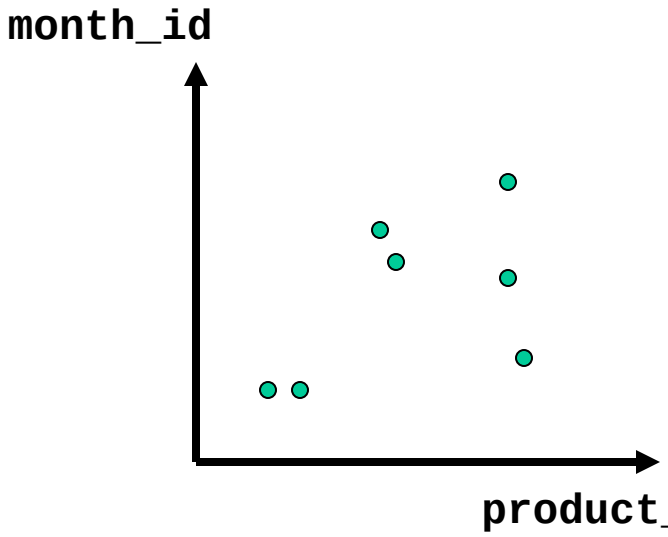
Content of this Lecture

- Multidimensional Indexing
- Grid-Files
- Kd-trees
- Multidimensional range queries on modern hardware

Multidimensional Queries (MDQ)

- Conditions on more than one dimension (=attribute)
  - Combined through AND (intersection) or OR (union)
- Partial queries: Conditions on some but not all dimensions
- A MDQ selects a sub-cube
  - 2D: “All beverage sales in March 2000“
  - 4D: “All beverage sales in 2000 in Berlin to male customers”

Composite Indexes

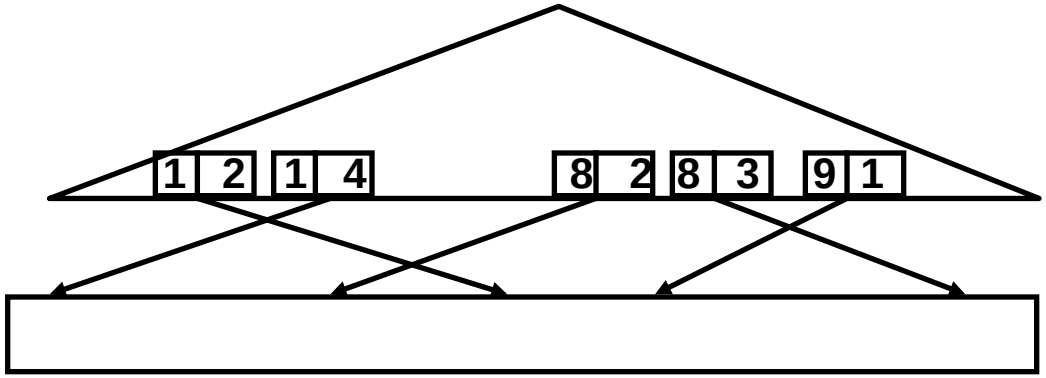


Point	X	Y
P1	2	2
P2	2	2
P3	5	7
P4	5	6
P5	8	6
P6	8	9
P7	9	3

- Imagine composite index on (X, Y)
- Efficiently supported
  - Full box queries (conditions in all dimensions X and Y)
  - Points/range with X between ...
- Not efficiently supported
  - Points/range with Y between ...

Composite Index

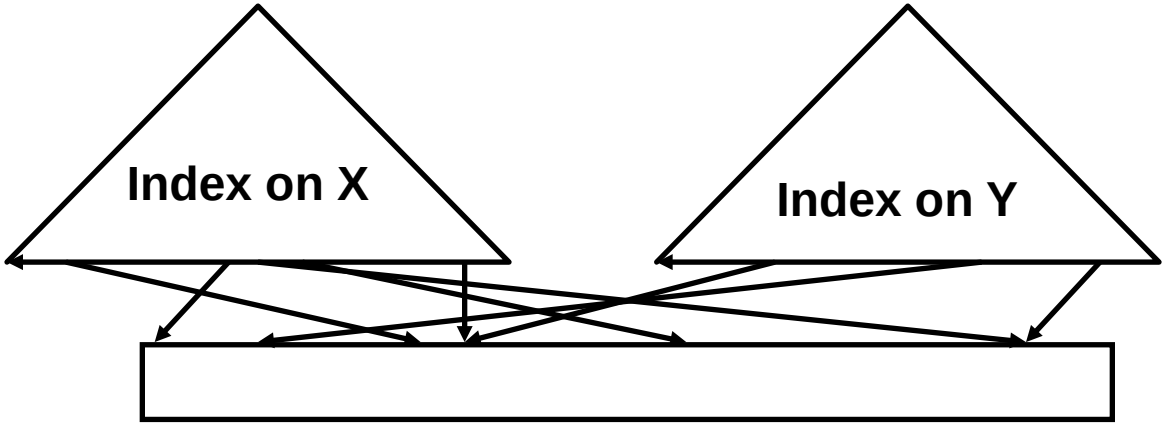
- One index over two concatenated attribute values (X, Y)



- For an concatenated index I to be eligible for a query Q, a prefix of the attributes of I must be present in Q
  - The longer the prefix in the query, the better
  - Better - higher the selectivity, more pruning
- Alternatives: Use independent indexes on each attribute

Independent Indexes

- One index per attribute



- Point/range query on one attribute: supported
- Point/range query on >1 attributes
  - Compute TID lists for each attribute
  - Intersect

Independent versus Composite Index

- Consider 3 dimensions of range 1,...,100
  - 1.000.000 points, uniformly distributed at random
  - Assume index blocks hold 50 keys or records
  - B*-Index on each attribute has height 4
- Find points with $40<x\leq 50$, $40<y\leq 50$, $40<z\leq 50$
- Using independent indexes
  - Using x-index, we generate TID list $|X|\sim 100.000$
  - Using y-index, we generate TID list $|Y|\sim 100.000$
  - Using z-index, we generate TID list $|Z|\sim 100.000$
  - For each index, we have $4+100.000/50=2004$ IO
    - Assumption: TIDs sorted in sequential blocks with 50 TIDs each
  - Hopefully, we can keep the three lists in main memory
  - Intersection yields ~1.000 points with 6012 IO

Independent versus Composite Index

- Consider 3 dimensions of range 1,...,100
  - 1.000.000 points, uniformly distributed at random
  - Assume index blocks hold 50 keys or records
  - B*-Index on each attribute has height 4
- Find points with $40<x\leq50$, $40<y\leq50$, $40<z\leq50$
- Using composite index (X,Y,Z)
  - Number of indexed points doesn't change
  - Key length increases – assume blocks hold only 30 (10) keys or records
  - Index has height 5 (6)
    - This is worst case – index blocks only 50% filled
  - Total: 5 (6) +1000/30 (10) ~38 IO (106)
    - Matching points are packed in a few blocks
    - This will be random access IO

Conclusion

- We want composite indexes
- Much less IO
- Things get worse for larger d
  - TID lists don't fit into main memory – paging, more IO
  - Intersecting many large TID lists can be more work than scanning all points once
- Advantage of composite indexes grows "exponentially" with number of dimensions and selectivity of selections
- Things get complicated if data is not uniformly distributed
  - Dependent attributes (age – weight, income, height, ...)
- But: For partial queries, we would need to index all combinations

Solutions

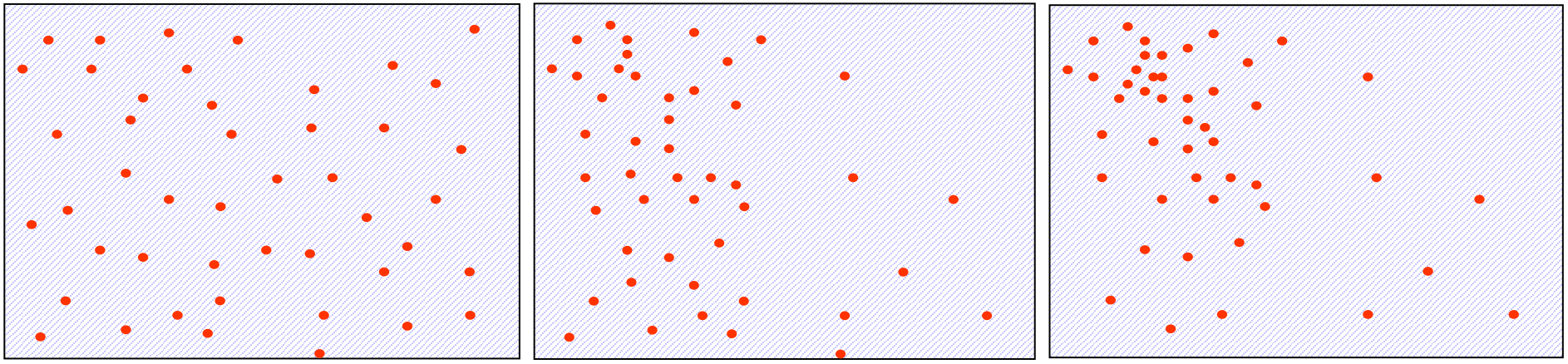
- One solution: Bitmap-Index
  - Bad choice if cardinality of attributes is high
  - Only point-queries are supported efficiently
  - “Read-only”, always needs to go back to the data files
- Other solution: Multidimensional index structures (MDIS)
  - Large improvements in principle
  - Advantages: Can grow/shrink; handle skew to some degree; nearest neighbor search
  - Made it into practice only for spatial data (small d)
- “Curse of dimensionality”: MDIS degrade for large d
  - Bad space usage, excessive management cost
  - Accesses degrade to sans

Multidimensional Indexes

- All dimensions are equally important
- Should support all types of queries
  - Exact match point queries, range queries
  - Partial match or range queries
  - Nearest neighbor queries (similarity search)
- Main trick: Try to store neighbors (in attribute space) in nearby storage locations (disk blocks, memory pages)
  - Translate locality in attribute space in locality in storage space
  - Difficult to achieve, key to good performance
- Why not B*-trees?
  - All B-trees need a total order on the keys
  - For more than one dimension, no 1D-order exists

Data Skew

- We say data is skewed if its values to not follow the expected distribution
- In MDIS, the typical expectation is uniform distribution
- Anything not uniform is (more or the less) skewed
- Data can be skewed in one or more dimensions



Content of this Lecture

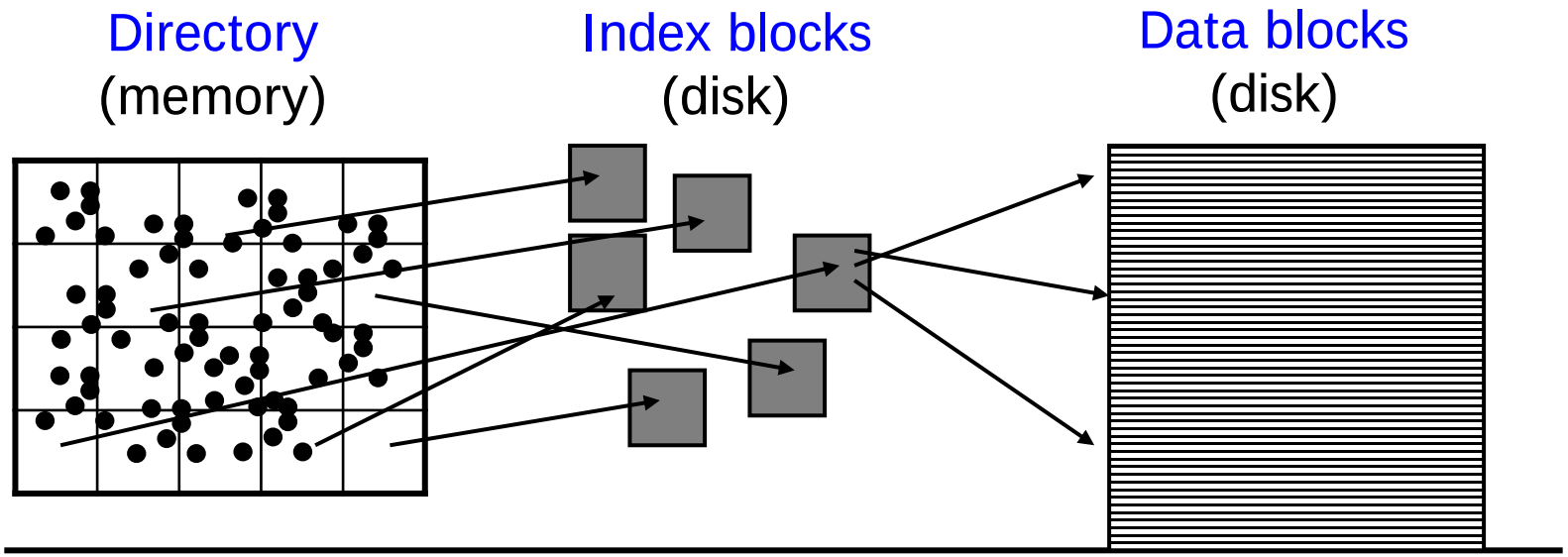
- Multidimensional Indexing
- Grid-Files
  - General Structure
  - Splits
- Kd-trees
- Multidimensional range queries on modern hardware

Grid-File

- Classical multidimensional index structure
  - Simple: searching, (inserting), ((deleting))
  - Good for uniformly distributed data
  - Does not handle skewed data very well
  - Many variations
- Design goals
  - Index structure for point objects
  - Support exact, partial match, range, and neighborhood queries
  - Guaranteed “two IO” access (under some assumptions)
  - All dimensions are treated the same
  - Adapts dynamically to the number of points

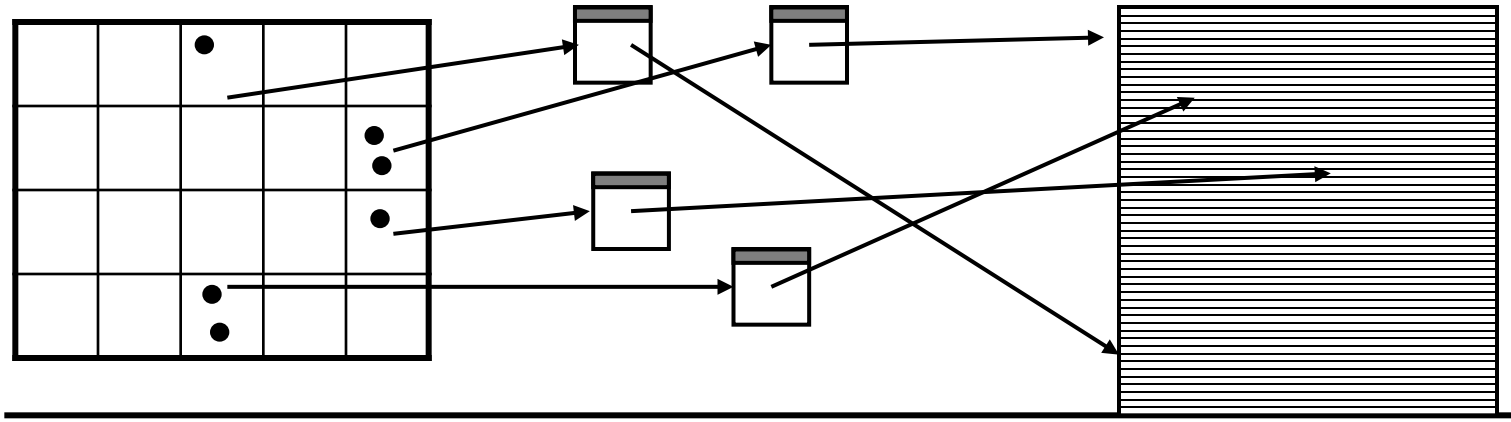
First Idea: Fixed Grid

- [Does not adapt to data distribution at all]
- Idea
  - Split space into equal-spaced cuboids or cells
    - We need maximal and minimal values for each dimensions
  - Directory stores one pointer to an index block for each cell
  - Index blocks: Points with coordinates and pointer to data record



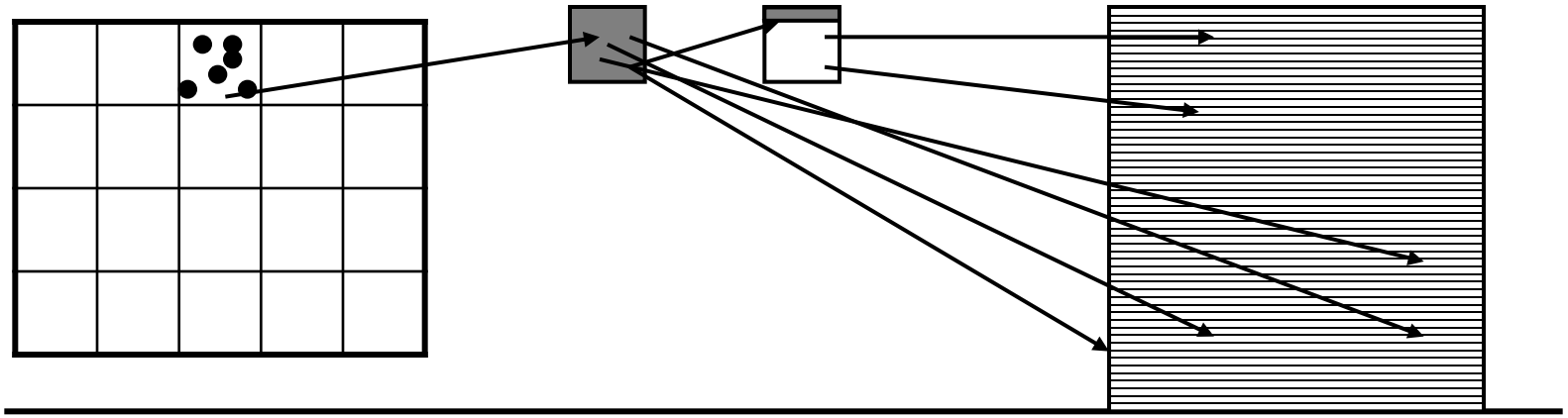
Operations

- Problem 1: Empty space
- Deleting a point
  - Compute cell using coordinates
  - Search cell in directory and load index block
  - Search point and delete, if present (also delete in data block)
  - Index block may become almost empty
- Index may consist of many almost empty index blocks
  - And how should we set the number of splits per dimension?



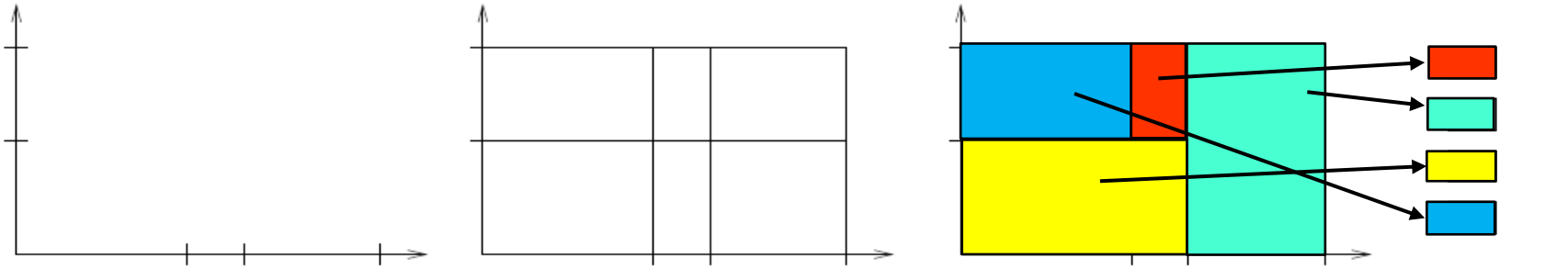
Operations

- Problem 2: Index blocks only hold a fixed # of pointers
- Inserting a point
  - Locate and load index block
  - If free space: insert point (also into data block)
  - If no free space: Generate overflow index blocks
  - No adaptation to skewed data distributions
  - Degenerates to a scan if all points fall in the same (set of) cells



Principle of Grid-Files

- Partition each dimension into disjoint intervals (scales)
  - Scales may be non-uniform and different for different dimensions
- Intersections of all intervals define all grid cells
  - d-dimensional cuboids
  - Each cell holds one pointer to the index block of the cell
  - Each point falls into exactly one grid cell
  - Many cells may point to same index block (less empty space)
  - When cell overflows – split cell (no overflow index blocks)



Exact Point Search

- Finding query point p (with full coordinates)
  - Keep scales for each dimension in memory
  - Look-up query coordinates in scales and derive grid cell
  - Extract pointer to index block from grid cell
  - Load index block and scan for p
- Complexity
  - We assume that the directory is in main memory
    - Other techniques exist, i.e., B*-tree over grid coordinates
  - Load index block (1st IO)
  - Search point in index block (no IO)
  - Access record following pointer (2nd IO)
  - Guaranteed 2 IO (two block random access)

Range Query, Partial Match Query

- Range query
  - Compute grid cell coordinates for each end point
  - All grid directory entries in that range may contain qualifying points
  - Extract all pointers to index blocks and scan
- Partial match query
  - Compute partial grid cell coordinates
  - All grid directory entries with these coordinates may contain points
  - Extract all pointers to index blocks and scan

Content of this Lecture

- Multidimensional Indexing
- Grid-Files
  - General Structure
  - Splits
- Kd-trees
- Multidimensional range queries on modern hardware

Inserting Points

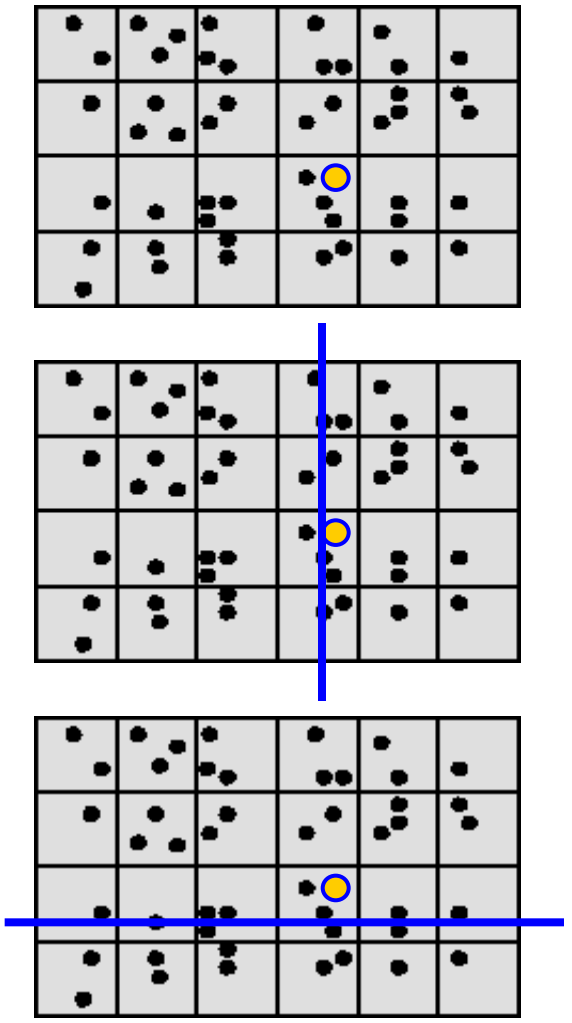
- If index block has free space – no problem
- Otherwise (1st option): Split cuboid at new scale
  - Choose a dimension and a scale to split
  - Create new scale, create new index block, distribute points in overflown block according to the chosen split
  - Insert point into matching index block
  - This implicitly splits all other grid cells with the same scale
  - All other cells: Copy pointer; old and new cells point to the same index block (only main memory work)

Choices

- Choice of dimension and scale to split is difficult
  - Optimally, we would like to split as many currently very full index blocks as evenly as possible
    - This is an optimization problem
  - We may also consider future insertions
    - Then we need formalized expectations (e.g. data distributions)

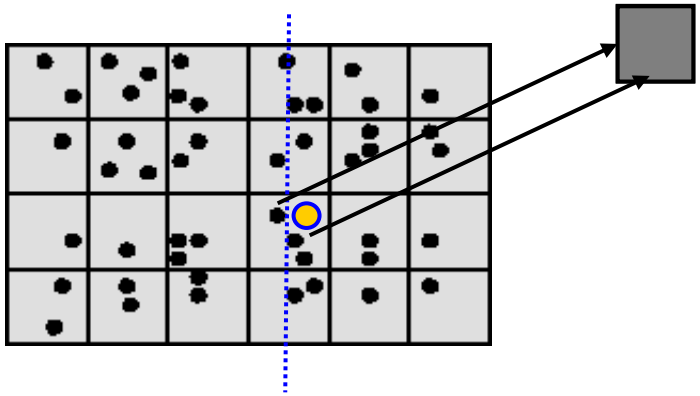
Example

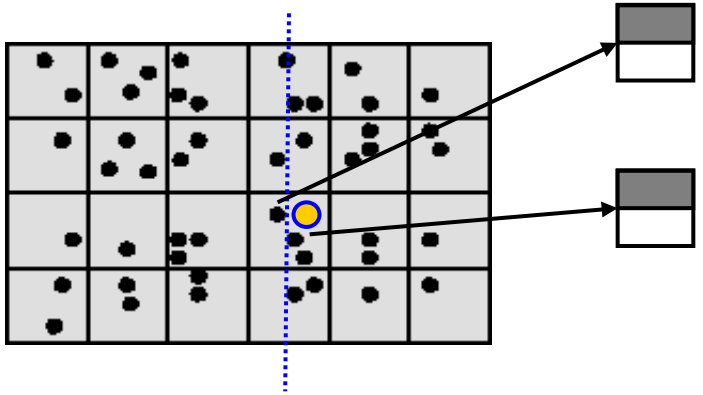
- Imagine block holds 3 pointers
  - Note: Usually we have unevenly spaced intervals
- New point causes overflow
- Vertical split
  - Splits 2 (3,4)-point blocks
  - Leaves one 3-point block
- Horizontal split
  - Splits 2 (3,4)-point blocks
  - Leaves one 3-point block
- Need to consider $O(k^{d-1})$ regions
  - Where k= # of scales per dimension
- Note: Those splits are not realized immediately on disk



Inserting Points -2-

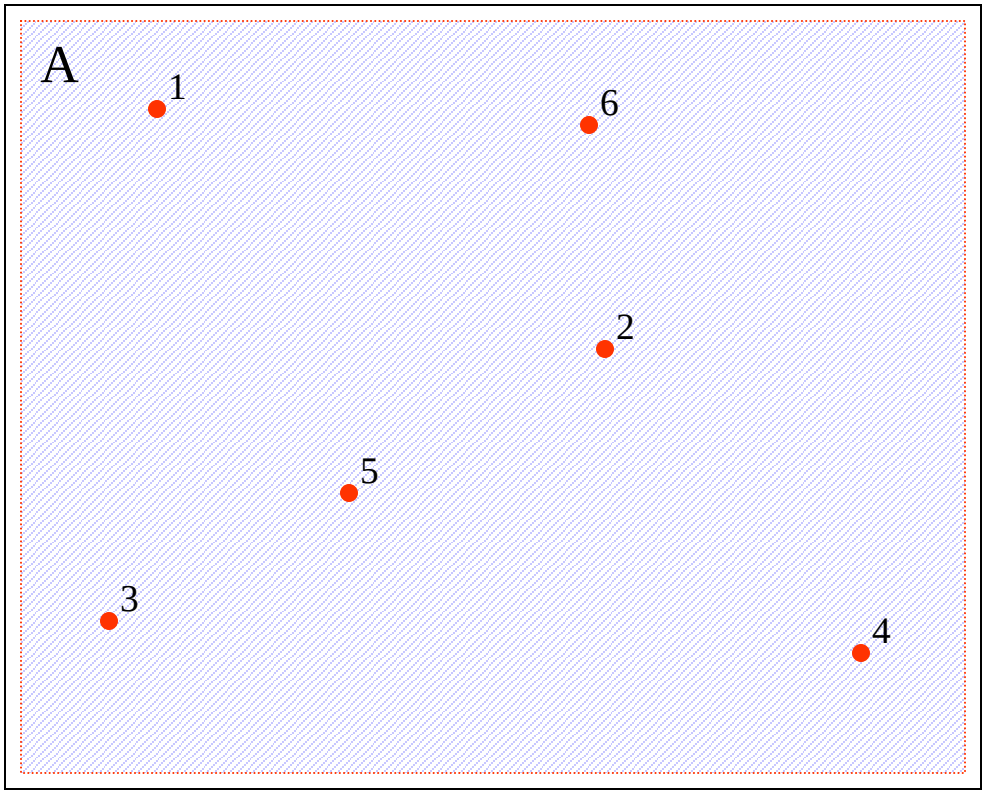
- 2nd option: If there are scales in the overflown region that do not yet have their own index pointers
  - Chose best such split, create new index block, distribute points, update pointer in grid cell
  - Other cells / blocks are not affected

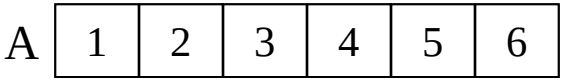




Grid-File Example 1 (from J. Gehrke)

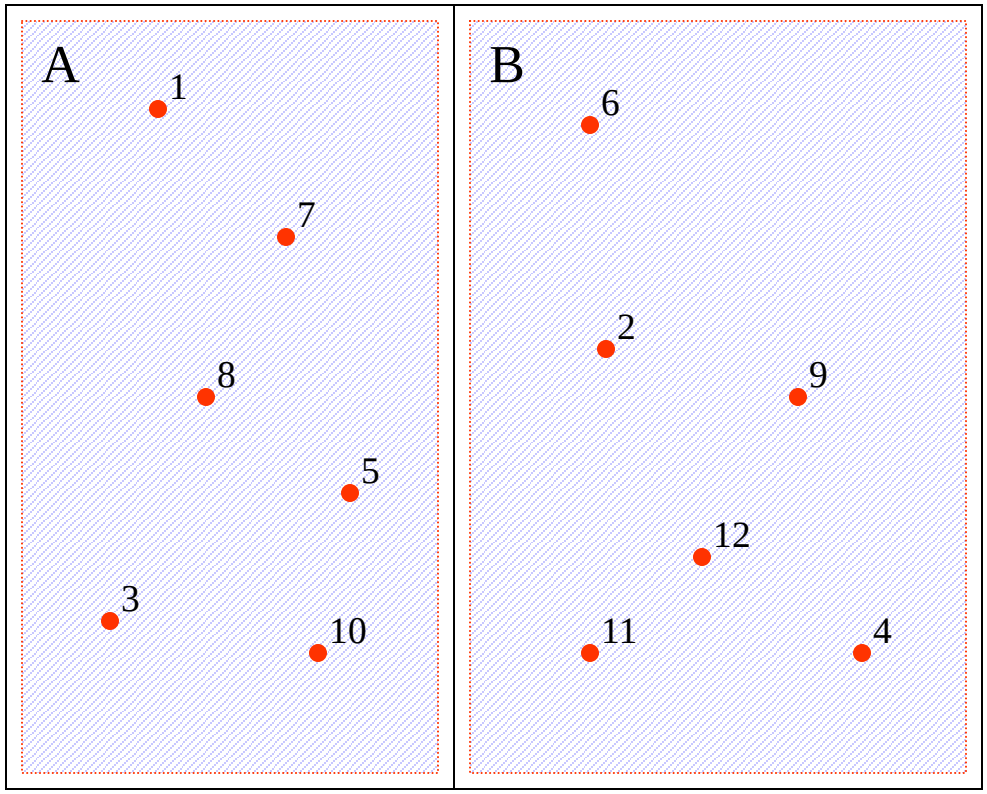
(N=6)





Grid-File Example 2

(N=6)

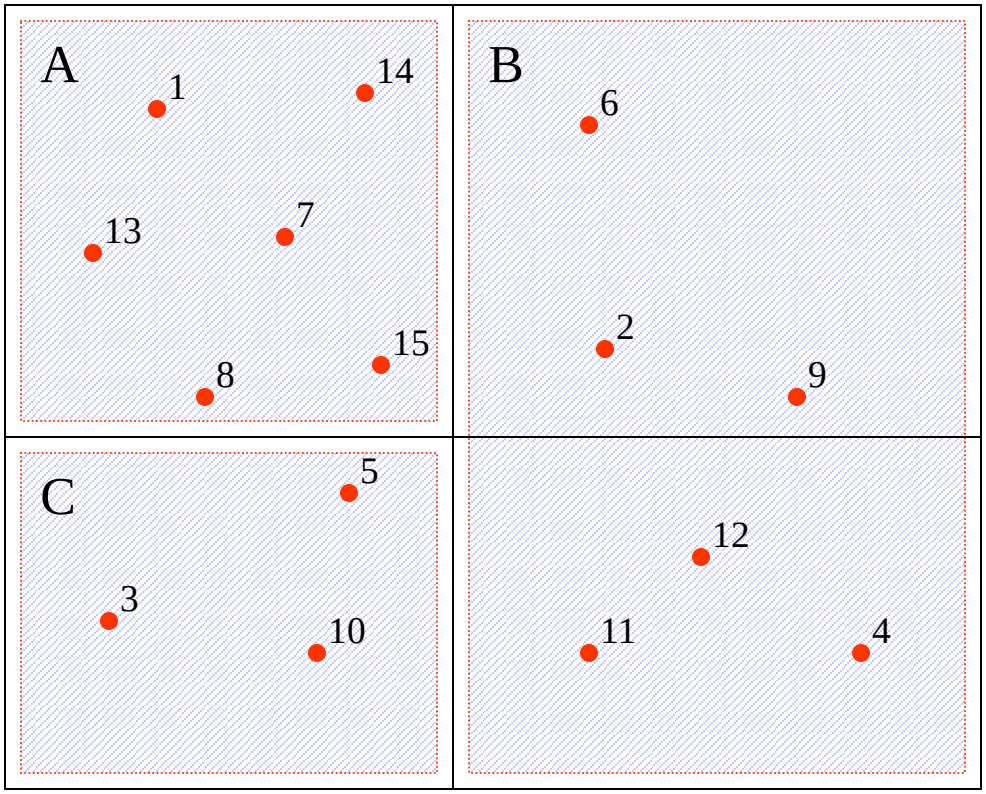


A	B

A	1	3	5	7	8	10
B	2	4	6	9	11	12

Grid-File Example 3

(N=6)

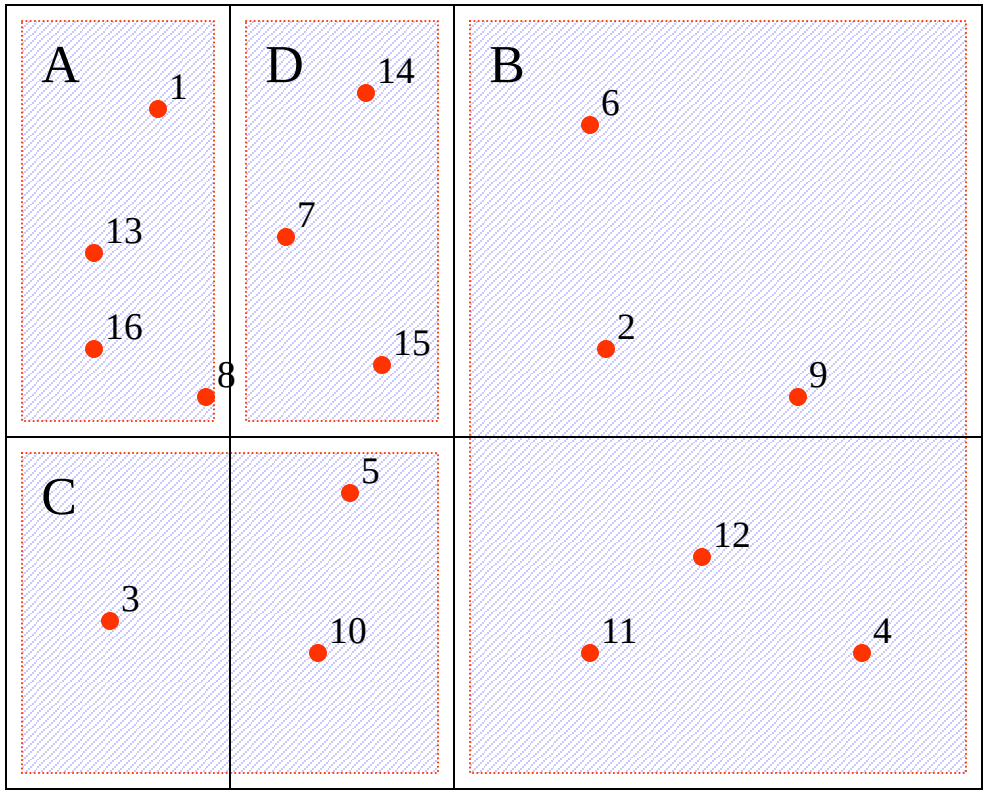


A	B
C	B

A	1	7	8	13	14	15
B	2	4	6	9	11	12
C	3	5	10			

Grid-File Example 4

(N=6)

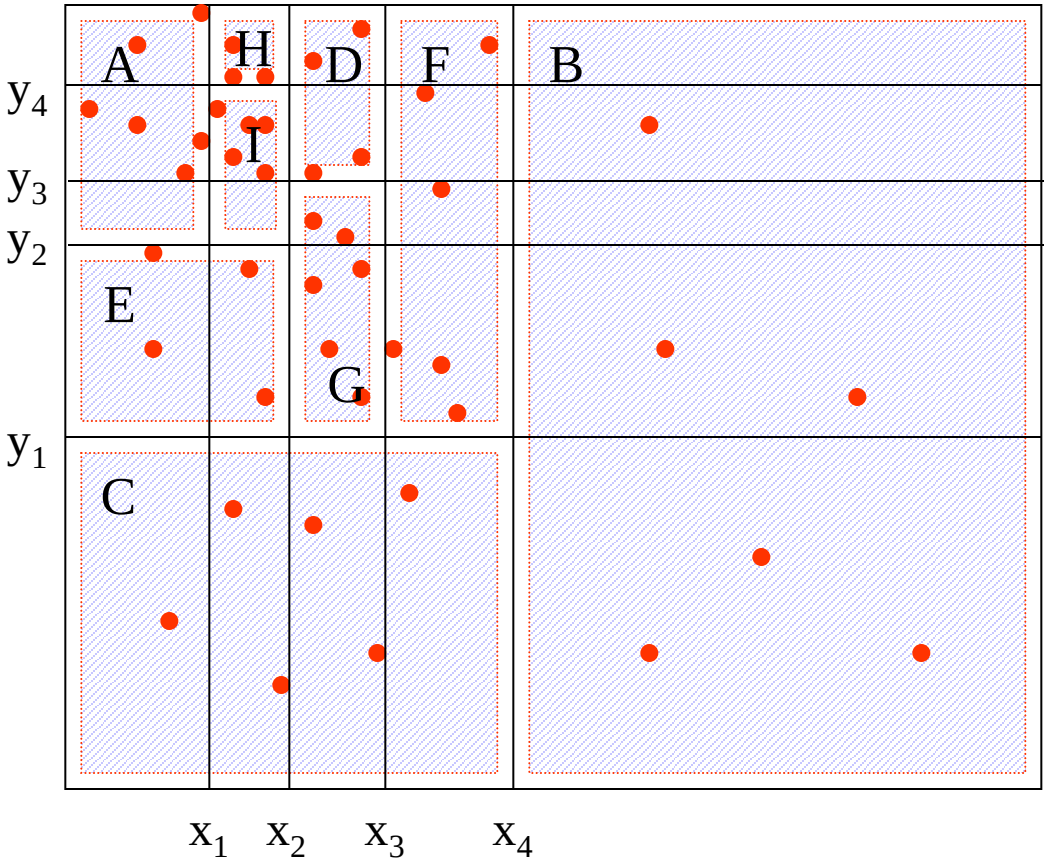


A	D	B
C	C	B

A	1	8	13	16		
B	2	4	6	9	11	12
C	3	5	10			
D	7	14	15			

Grid-File Example 5

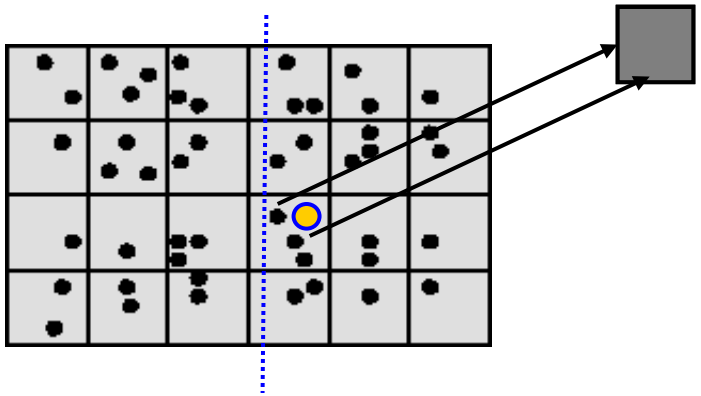
(N=6)

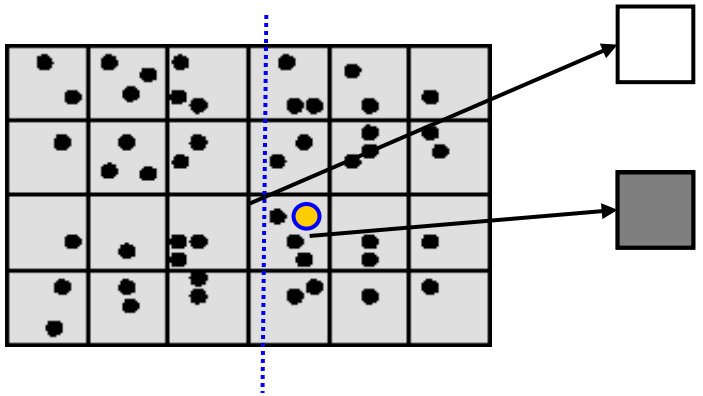


A	H	D	F	B
A	I	D	F	B
A	I	G	F	B
E	E	G	F	B
C	C	C	C	B

Problems

- What if un-realized scales does not lead to even distribution of points during a split?
  - New splits are created based on a local decision (the overflown region) and on past data
  - But they influence other cells in the future
- Actually, we should split such that all affected cells are evenly distributed in the future – but we cannot



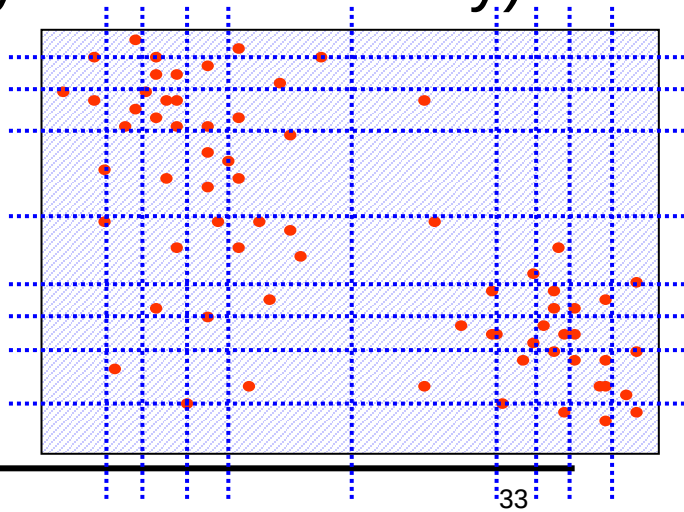


Deleting Points

- Search point and delete
- If index block become “almost” empty, merge blocks
  - A merge is the removal of a split
  - All other almost empty index blocks are candidates for merging
  - A merge should build a convex region
    - Or range queries need to look into unnecessarily many blocks
  - This can become very difficult
    - Potentially, more than two regions need to be merged to keep convexity condition
- No details here

Conclusions

- Grid-Files always split parallel to the dimension axes
  - This is not always optimal
  - Use others than rectangles as cells: circles, polygons, etc.
    - Might not disjointly fill the space any more
    - Allow overlaps - R trees
- Good: Good index block fill degrees if distribution of points does not change over time
- Good: Two IO guarantee (if directory fits into memory)
- Bad: Grid directory grows very fast
- Bad: Bad adaptation to “unevenly skewed” data
  - The more dimensions, the worse



Content of this Lecture

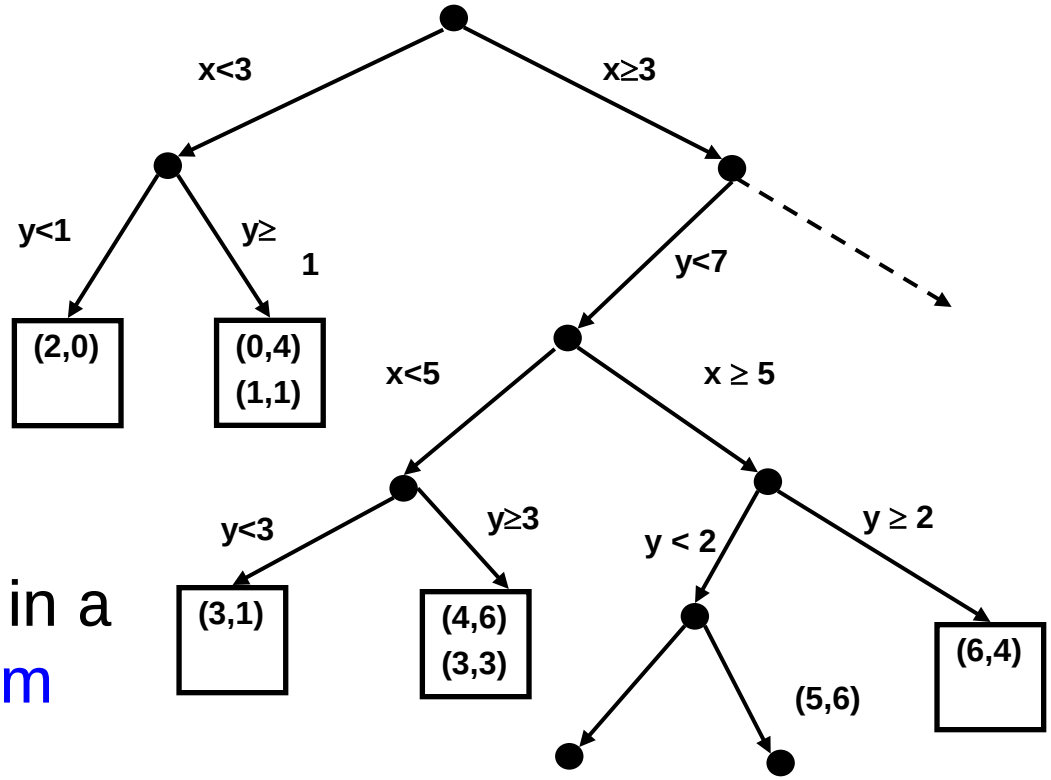
- Multidimensional Indexing
- Grid-Files
- kd-trees
- Multidimensional range queries on modern hardware

kd-tree

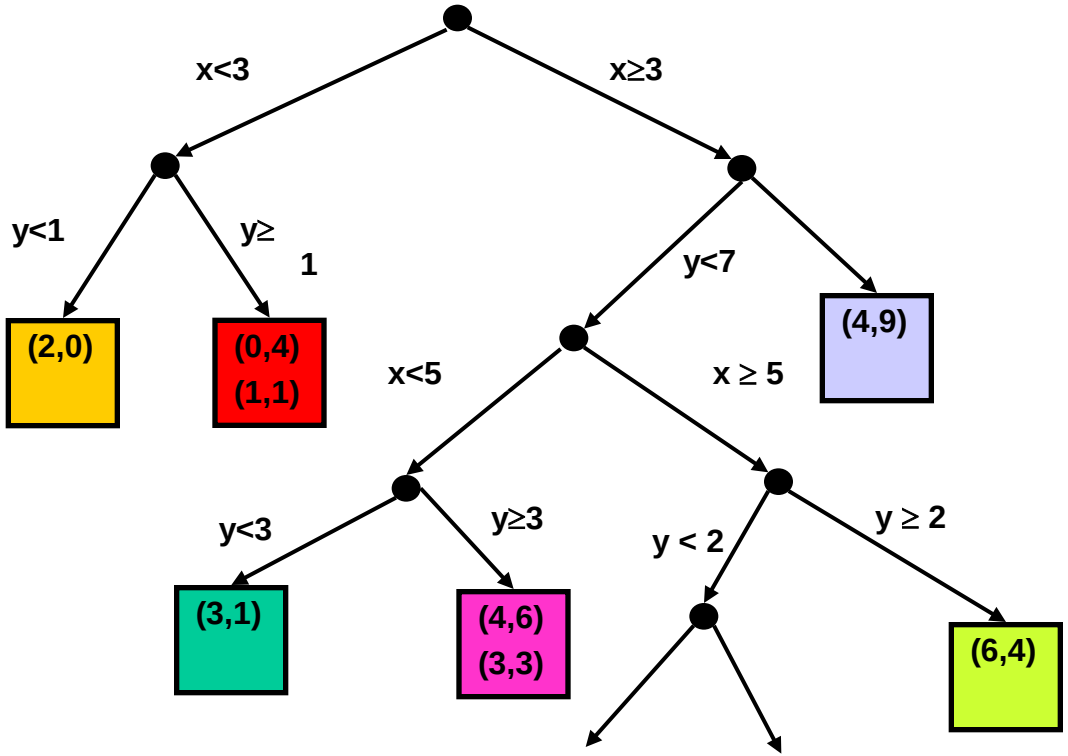
- Grid-File disadvantages
  - All regions of the d-dimensional space are eventually split at the same dimension / scale
  - First cell that overflows determines split
  - This choice is global and never undone
- kd-trees
  - Multidimensional variation of binary search trees
  - Hierarchical splitting of space into regions
  - Regions in different subtrees may use different split positions
  - Better adaptation to clustered data than Grid-Files
  - kd-tree originally is a main memory data structure

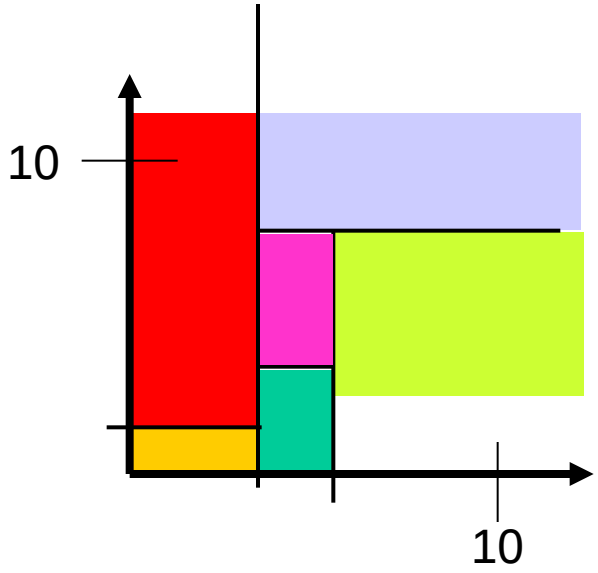
General Idea

- Binary, rooted tree
- Paths are selected by dimension / value
- Dimensions are not statically assigned to levels of the tree
- Data points are stored only in leaves
- A leaf stores all points in a n-dim hypercube with m border planes ($m \leq n$)
- Leaves are stored on disk

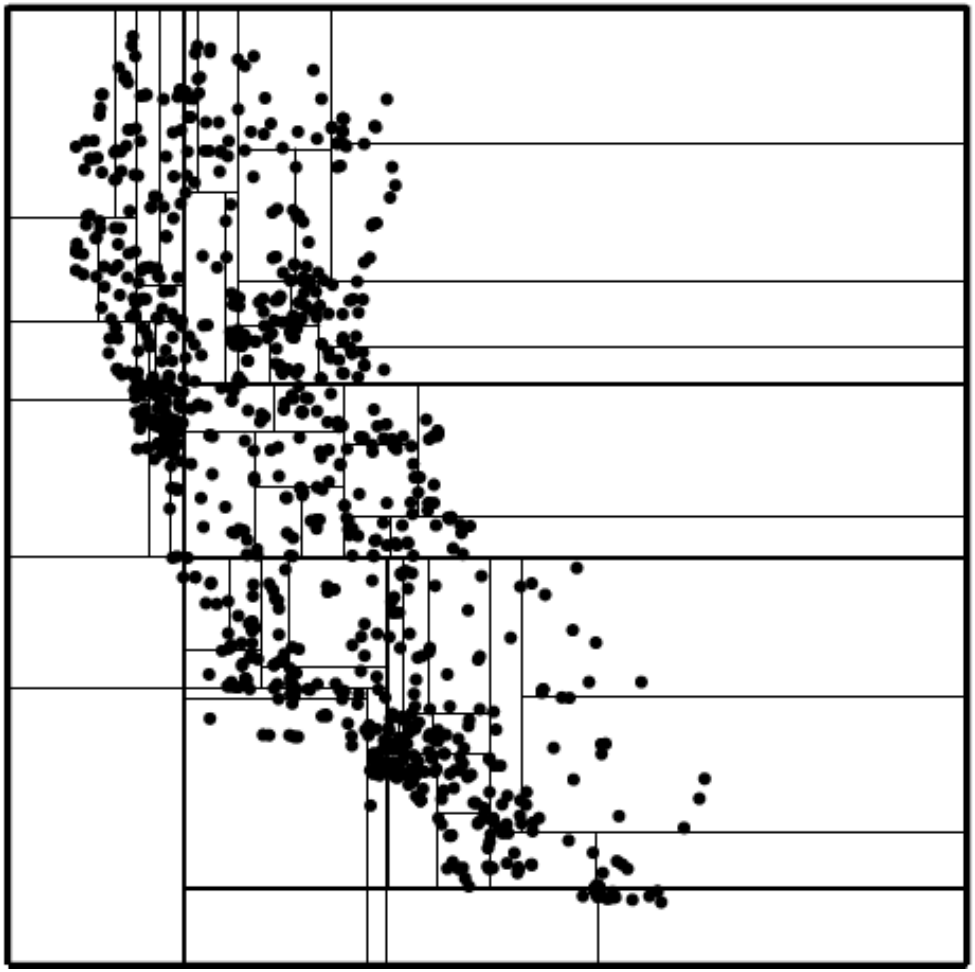


Example – the Brick wall





Local Adaptation



Search Operations

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

Search Operations

- Exact point search
  - In each inner node, decide direction based on split condition
  - Search leaf for query point
  - Complexity depends on depth of leaf
  - kd-Trees are not balanced
    - No guarantees (except data set size)
  - Only leaves are on disk – 1 IO to obtain TIDs
- Range Query
  - Follow all children which might have points within the range
  - Need for multiple search paths
- Partial match query
  - ...

kd-tree Insertion

- Find appropriate leaf block
- If free space available – insert, done
- Otherwise, chose split dimension and position
  - This is a local decision; remains stable for the future of the subtree
  - Find dimension and split that divides set of points into two sets
    - Consider current points and split in sets of approximately equal size
    - Consider known distributions of values in different dimensions
    - Use alternation scheme for 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 lead to tree degradation
  - CS students at HU: Don’t split at sex, place of birth, ...

Summary

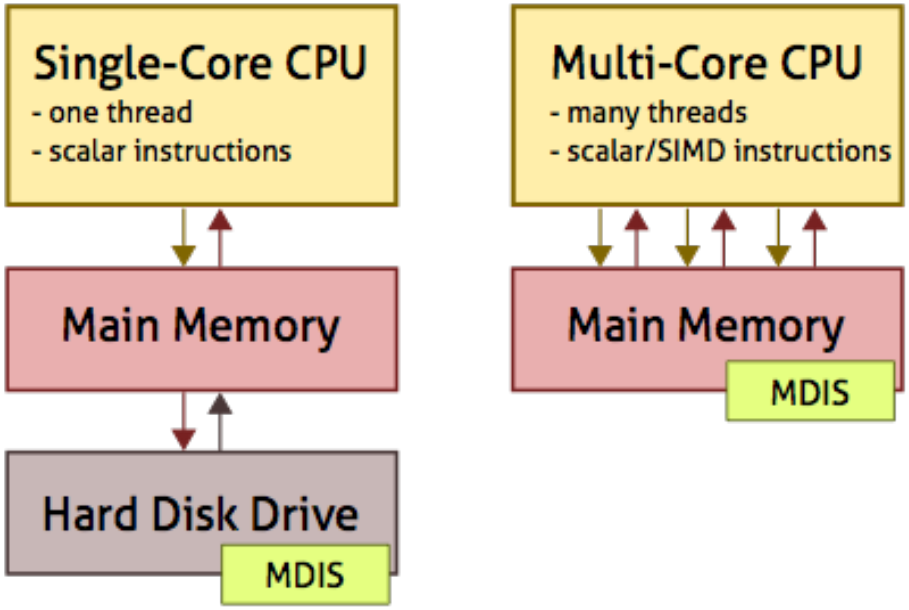
- We gave an overview on MDIS
- Other MDIS's: Partitioned hashing, R-tree, Quad-Tree, X tree, hb tree, R+ tree, UB tree, ...
  - Store objects more than once; other than rectangular cells; spatial objects; ...
- Not discussed: Similarity search
- Curse of dimensionality
  - The more dimensions, the more difficult to manage an MDIS
    - Grid-File: Every split creates exponentially many more cells
    - Kd-Tree: Which dimension to chose for next split
  - Often, linear scanning of objects is quicker
    - When is "often"?

Content of this Lecture

- Multidimensional Indexing
- Grid-Files
- Kd-trees
- Multidimensional range queries on modern hardware

Scan or Index?

- "Hence, we consider an index structure to work ‘well’ if, on average, less than 20% of blocks must be visited, and to ‘fail’ if, on average, more than 20% of blocks must be visited."
  - Weber et al., VLDB 1998
  - 20%? Assumption: Scanning is ~5 times faster than random access
  - MDIS always good for exact queries, but range queries?
- Does large memory and multi-core change the game?

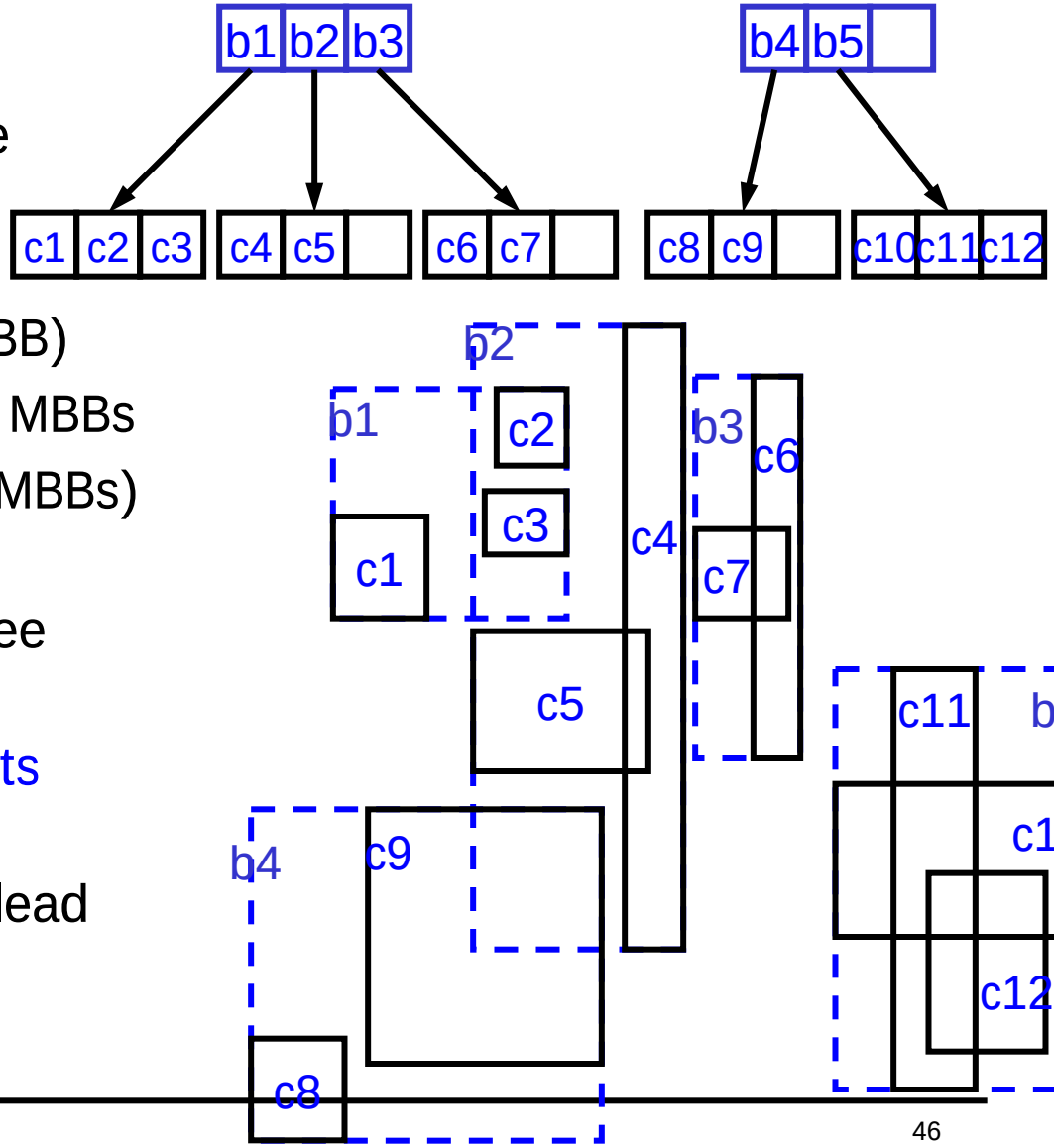


Benchmark

- Three popular MDIS and two flavors of linear scans
  - R-Tree, kd-Tree, VA-File
- Four different data sets
  - Two real: 3D sensor data, 19D genomics data
  - Two synthetic: Uniform, with multiple sub-clusters
- Synthetic workloads of range queries with varying selectivities
- All methods are parallelized and use SIMD
- Measurements on multi-core server with 1TB RAM
- [work by Stefan Sprenger (submitted)]

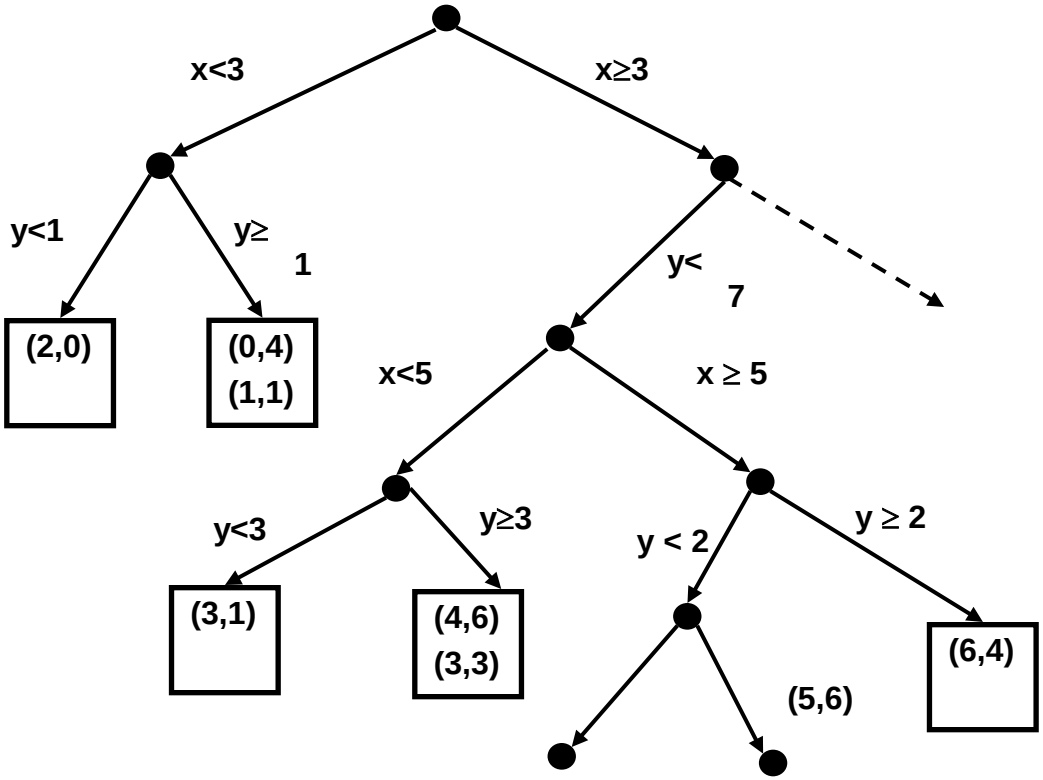
R-Tree

- Hierarchical data structure
- All points in a node are represented by their minimal bounding box (MBB)
- Inner nodes hold multiple MBBs
- On overflow, blocks (and MBBs) are split
- Splits propagate up the tree
  - R-tree is balanced
- Designed for spatial objects
  - MBB may overlap
- Even point searches may lead to multiple search paths



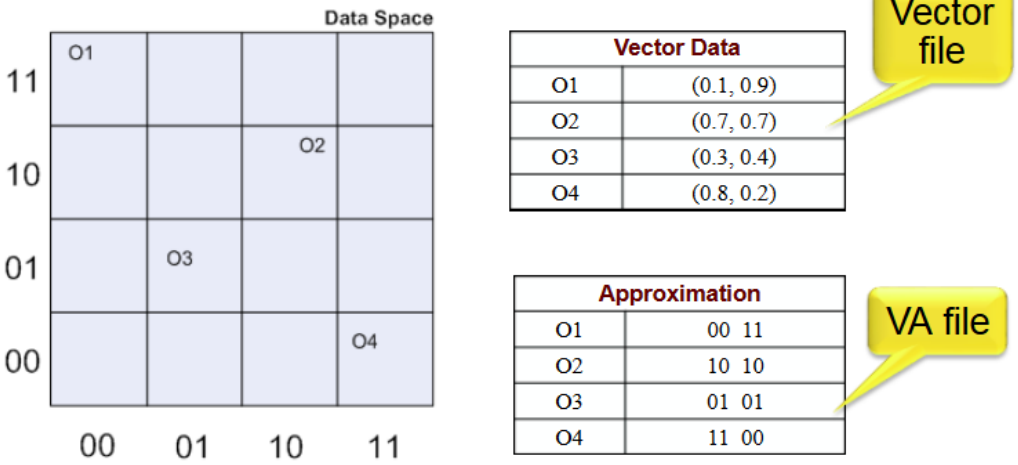
kd-Tree

- Unbalanced
  - We inserts tuples in random order, which creates almost balanced trees
- Split dimension selected by round-robin
- Leaf sizes adapted to cache lines



VA-File

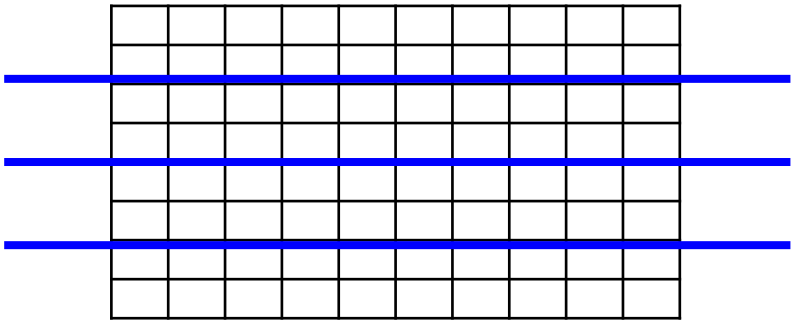
- Similar to a GRID file
- Partition each dimension into equi-distance bins
- Bins are addresses using equal-size bitstrings
- „Approximate“ address of an object is a m-part bitstring (m: Dimensions)
- Each bitstring value addresses a data block
- Upon a range query, all matching bit-strings are determined and data blocks scanned
- (Not adaptive at all)



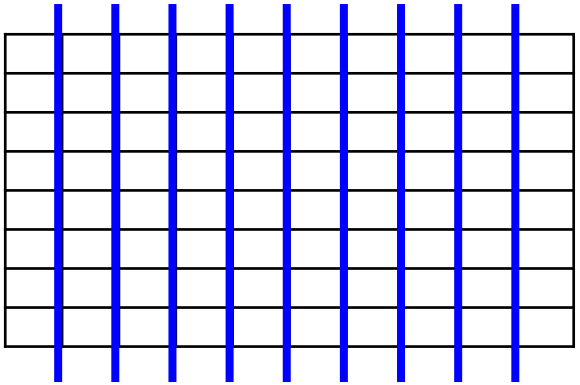
Source: D. Lamb, “Search Techniques for Multimedia Databases”

Parallelization by Partitioning

- Most parallelization techniques build on partitioning the data into p disjoint partitions
- Let D be the set of points, each having k dimensions
- Horizontal partitioning
  - Partitions contain |D|/p points
  - We only use random partitioning
    - No locality
    - Would be very difficult to keep in case of updates

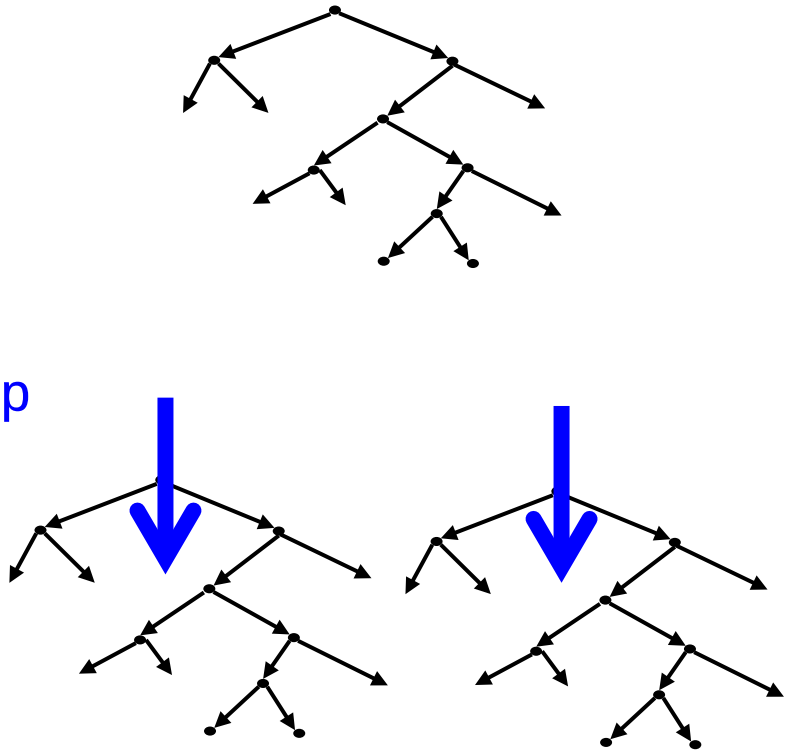


- Vertical partitioning
  - Only possible if k=p
  - Each partition contains one attribute of all |D| points



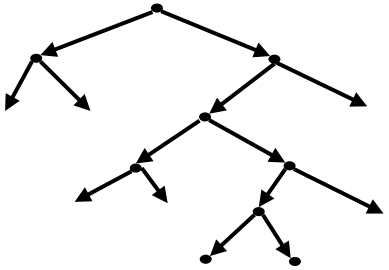
Searching a Tree in Parallel

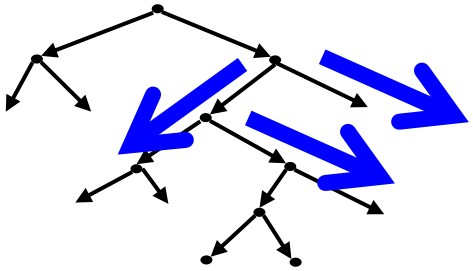
- How to parallelize search through a tree over a large set of points?
- Option 1: Partition data set
  - Partition data set horizontally into p partitions
    - Usually p=t, # of threads
  - Build one tree per partition
  - Given a query, all p trees are traversed in parallel
  - Union of partitioned results gives final result



Searching a Tree in Parallel

- How to parallelize search through a tree over a large set of points?
- Option 2: Parallel traversal
  - Build tree over entire data set
  - Given a query, traverse the tree
  - Whenever a parallel search path emerges, span a new thread
  - When more paths emerge than t – start scheduling
  - Different threads create independent results – union gives final result





Parallelization

- R-Tree, kd-Tree, VA File
  - Partition data horizontally and build one tree per partition
  - Given a query, all trees are traversed in parallel
  - Number of partitions = number of threads
  - SIMD is used in leaf nodes
- Scan
  - Horizontal partitioning: All partitions are scanned by one thread in parallel
  - Vertical partitioning: Scan each partition in parallel and with SIMD and build a bitmap flagging matching tuples
  - Use fast bit operations to intersect all bitmaps for final result
  - Very effective for partial match queries
    - Only a fraction of data is touched
    - But low degree-of-parallelism: # threads used = # dims in query

Results

- Randomly generated all-dim MDRQ executed over 1M uniformly distributed 20-dimensional integer vectors
- 24 CPU threads, 256-bit wide SIMD registers
- SIMD does not yield much benefit
  - Neither for single nor for multi threaded implementations
- Scans are faster despite high selectivity
  - On uniformly distributed data
- VA-file beats other MDIS
  - On uniformly distributed data

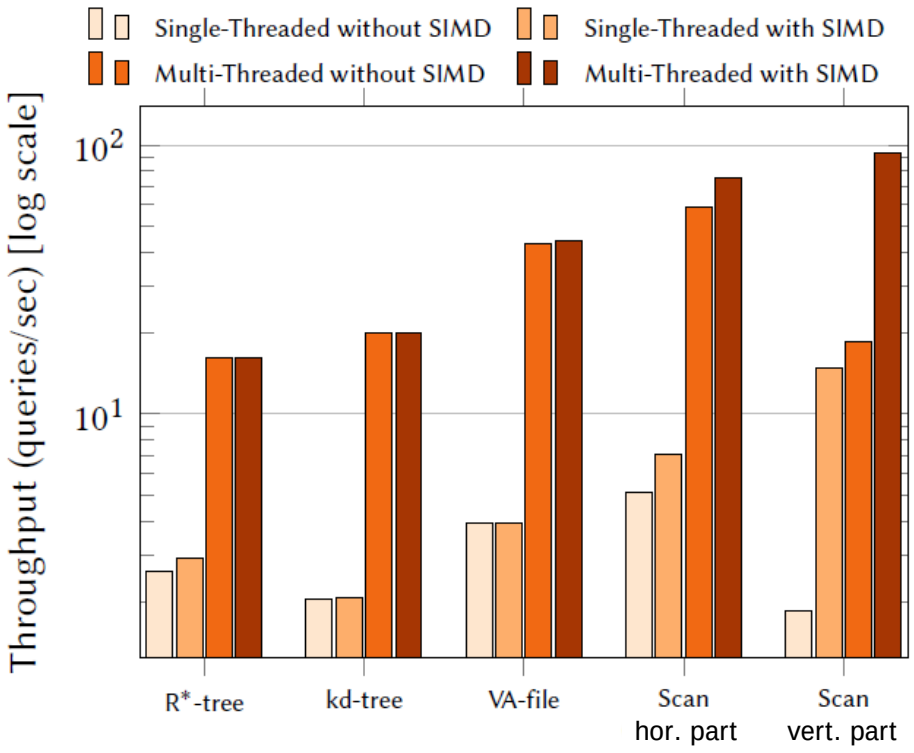


Figure 6: Throughput when executing MDRQ with an average selectivity of 0.1% on 1M 20-dimensional data objects depending on the used hardware features.

Dimensions, Selectivity

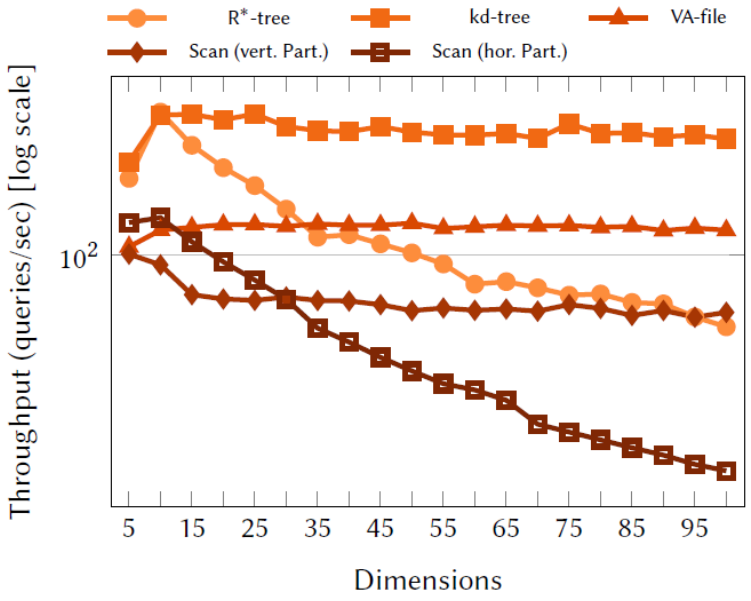


Figure 7: Throughput when executing range queries with an average selectivity of 0.4% (5 dimensions) to 0.0002% (more than 10 dimensions) on 1M uniformly distributed data objects depending on dimensionality.

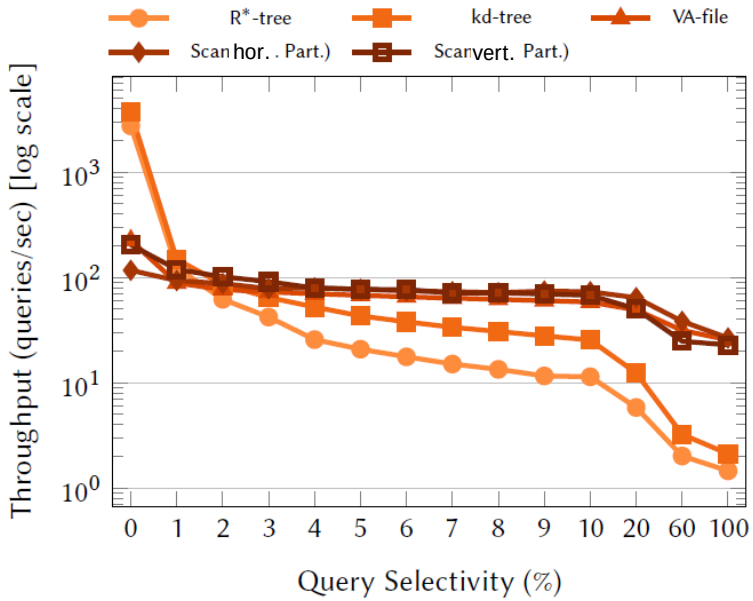


Figure 8: Throughput when executing range queries on 1M 5-dimensional uniformly distributed data objects using 24 software threads depending on query selectivity.

Real data, realistic workloads

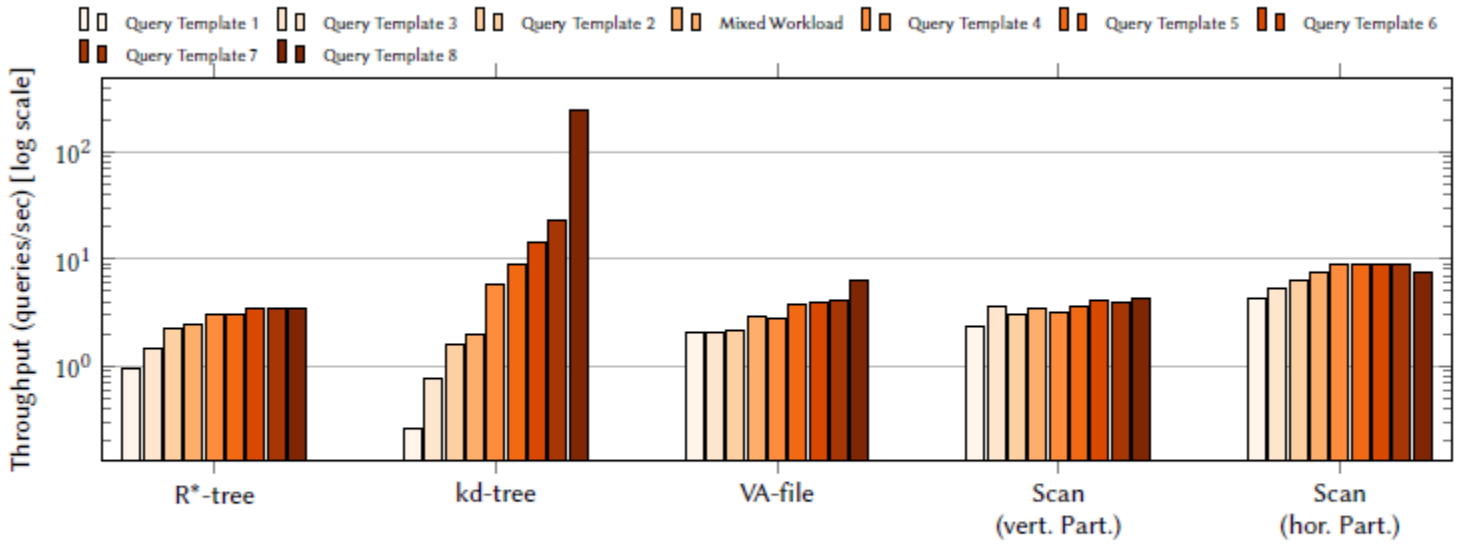


Figure 12: Throughput of contestants when executing the GMRQB with varying selectivities (see Figure 4) on 10M 19-dimensional data objects from the 1000 Genomes Project dataset using 24 software threads (query templates are ordered by average selectivity, from low to high).

- Scans are faster up to very high selectivities
- Of course, MDI could be better tuned to modern hardware

Literatur

- Gaede, V. and Günther, O. (1998). "Multidimensional Access Methods." ACM Computing Surveys 30(2): 170-231.
- Bentley, Jon Louis. "Multidimensional binary search trees used for associative searching." Communications of the ACM, 1975
- Nievergelt, Hinterberger, and Sevcik. "The grid file: An adaptable, symmetric multikey file structure." ACM TODS, 1984

Selbsttest

- How does a Grid File work?
- Static Grid files suffer from many empty blocks. How is solved in the original Grid file?
- If an index block overflows, a new scale is introduced. How much does this increase the size of the directory?
- What are strategies for choosing a split dimension in a kd-Tree?
- Why did we argue that a kd-Tree is a main memory data structure?
- What are typical properties of “modern hardware”,, and how do they change database architectures?