Data Warehousing und Data Mining

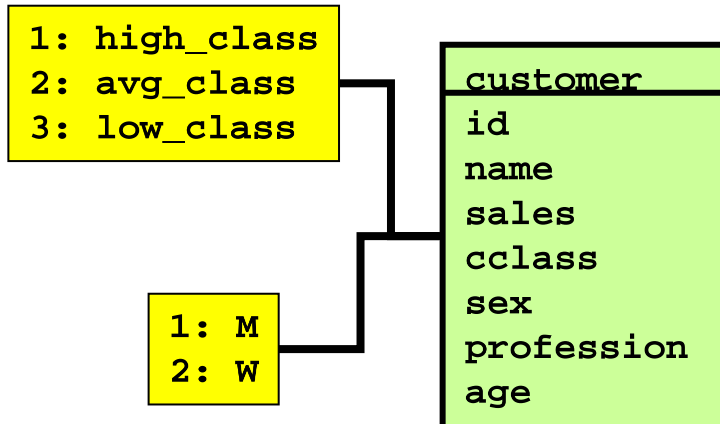
Multidimensionale Indexstrukturen

Ulf Leser

Wissensmanagement in der Bioinformatik

Grundaufbau

- Tabelle T mit Attribut A, |T|=n, |A|=a verschiedene Werte
- Repräsentation jedes der a Werte von A als Bitarray der Länge n
  - Attribut X, Wert z: X.z[i]=1 gdw i.tes Tupel hat Wert z in Attribut X
  - Die Ordnung der Tupel muss feststehen (später mehr)
- Repräsentation von A: Bitmatrix mit n*a Bits



Tabelle

```
22:45,1,Meier,20.000,2,M,...
A2:55,2,Müller,15.000,3,W,...
33:D1,3,Schmidt,25.000,1,M,...
1A:0E,4,Dehnert,22.000,2,M,...
...
```

Bitmap-Index `cclass`?

```
1:0010...
2:1001...
3:0100...
```

Bitmap-Index `sex`?

```
M:1011...
W:0100...
```

Vorteil 1 - Speicherplatz

- Vergleich Bitmap mit B*-Baum
- Kompakte Repräsentation bei kleinem a
  - Annahmen
    - n=1.000.000, zwei Attribute mit $a_1=4$, $a_2=2$, TID=4 Byte
  - 2 B*-Bäume
    - 2*4*1.000.000 = 8MB
    - In jedem Index ist jede TID einmal repräsentiert
    - Speicherplatz unabhängig von Kardinalität der Attribute
  - Bitmap-Index
    - 1.000.000 * (4+2)/8 = 0,75MB
- Vorteile
  - Kleiner Index passt eher in Hauptspeicher
  - Kleine Indexe können schneller von Platte gelesen werden

Vorteil 2 – Komplexe Bedingungen

- Bedingungen an mehrere Attribute
  - B*-Index
    - Lesen der TID Liste zu jeder Bedingung
    - Schnittmengenbildung aller Listen durch Sortierung oder Hashing
      - Sehr teuer bei geringer Selektivität
    - Auf Tupel in Schnittmenge zugreifen
  - Bitmap-Index
    - Lesen der Bitarrays für jede Bedingung
    - OR/AND Verknüpfung
      `„... cclass=3 AND sex=‚m‘ ...“` ⇒ `B[2]` ∧ `B[4]`
    - Auf Tupel mit passenden Bits zugreifen
- Kein wesentlicher Gewinn bei geringer Selektivität
  - Auch durch Bitmap-Indexe erhält man nur eine lange Liste von TIDs
- Großer Gewinn bei hoher Selektivität
  - Bitoperationen sehr schnell
  - Bitmatrizen sehr klein (viel kleiner als B*-Bäume)
- Außerdem: Ordnung der Attribute im Index ist egal

Run-Length-Encoding (RLE)

- RLE1: Explizites Speichern der 1-Positionen
  - Beispiel: n=1.000.000, a=100, TID: 4 Byte
  - Pro Bitarray ist nur einer von 100 Werten eine „1“
    - Annahme: Gleichverteilung der Werte von A über T
  - Bitmap ohne RLE: 1.000.000 * 100/8 = 12,5 MB
  - Bitmap mit RLE
    - 1.000.000 ist durch 20 Bit adressierbar
    - 1.000.000 * (20/8) /100*100 = 2.5MB
- Nachteil
  - Wir müssen die Größe des Arrays von vorneherein festlegen
    - Um die Anzahl der Bits für Kodierung festlegen zu können
  - Wenig adaptiv für schrumpfende / wachsende Tabellen

Kodierung

- Wir merken uns auch die Länge des Bitstrings für jede Blocklänge
  - Blocklänge benötigt x Bits – (x-1) Einser, dann eine Null
- Damit
  - 000101 ergibt Blöcke 3,1 ergibt 101101
  - 010001 ergibt Blöcke 1,3 ergibt 011011
  - 010101 ergibt Blöcke 1,1,1 ergibt 010101
  - Das sieht nicht nach Komprimierung aus!
- 1-er müssen selten sein
  - 1000000010000100
    - Blöcke 0,7,4 ergibt 00110111110100
    - Blocklänge 0 wird speziell als 00 kodiert
  - 0000000000100001
    - Blöcke 10,4 ergibt 1110101110100
- Noch seltener – 1 aus 100
  - 000000001000000000000000000000000000000000000000000000000100 ergibt 8,21 ergibt 11101011111010101

Vertikale Komprimierung

- Attribut A mit |A|=a verschiedenen Werten

t_1 t_2 t_3

```
1:00100000001010100000001010000…
2:10011100000010000110000000010…
3:01000000001000000000000100000…
…
a: …
```

- Komprimierung bisher

```
1:14,33,45…
2:3,55, …
3:17,34,41,…
…
a: …
```

- Können wir auch vertikal komprimieren (b<a)?

t_1 t_2 t_3

```
1:00100110001010100000001010011 0…
…
b: …
```

Verwendung anderer Zahlenbasen

- Beobachtung: a Werte
  - Binärkodierung: $\log_2(a)$ Bit
  - Bitmap: a Bit
- Gesucht seien alle Tupel mit A=x (oder A in {...})
- Vorteile Bitmapdarstellung
  - Jeder Wert „x“ entspricht genau einem Bitarray
  - Finden aller Tupel verlangt das Lesen eines Bitarrays
  - Zusammengesetzte Bedingungen durch logische Operationen auf Bitarrays
- Nachteil: Hoher Speicherbrauch (bei großem a)
- Kann man Kompromisse finden?
  - Ziel: Weniger Speicherverbrauch ohne viel mehr Bitarrays lesen zu müssen
  - Idee: Verwendung unterschiedlicher Zahlenbasen

Idee

- Repräsentation des Wertes x eines Tupels t nicht mehr durch a Bit, sondern in Bitmapdarstellung zu einer anderen Basis
- Beispiel: a=20, $t_1[a]=1$, $t_2[a]=18$, $t_3[a]=12$, $t_4[a]=11$, ...
  - Bitmap

```
01:1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
…
11:0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
12:0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
…
18:0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
19:0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
20:0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

  - Zur Basis <2,4,3>

```
 1 = 001 = 0*12 + 0*3 + 1
11 = 032 = 0*12 + 3*3 + 2
12 = 100 = 1*12 + 0*3 + 0
18 = 120 = 1*12 + 2*3 + 0
```

```
0:1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1:0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0:1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1:0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
2:0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
3:0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0:0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1:1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
2:0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Ergebnis

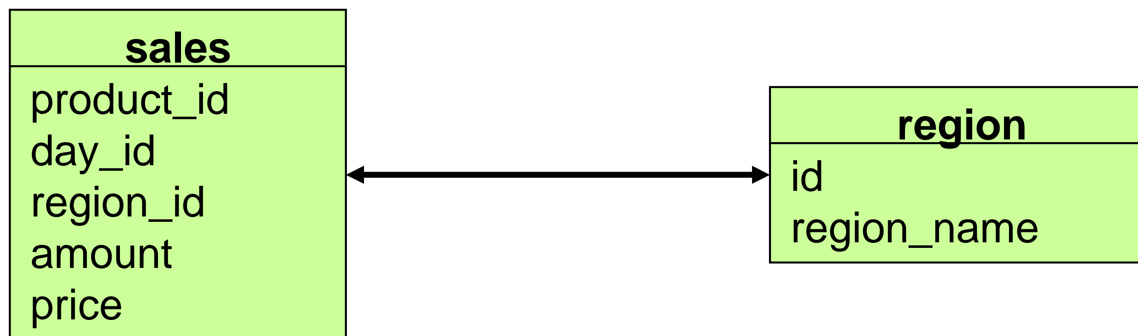
- Bitarrays sind gleich lang (im Unterschied zu RLE1/2)
- Aber man braucht weniger davon
- Das kostet bei Anfragen
  - Finden aller Tupel mit A=x benötigt Laden mehrerer Bitarrays
- Im Beispiel (<2,4,3>):
  - Platzverbrauch 9*n statt 20*n
  - Suche nach allen Tupel mit A=15: Lesen von 3 Bitarrays statt einem
- Implementierung
  - Lesen aller notwendigen Bitarrays
  - Logisches AND ergibt Bitarray mit allen Treffen
  - Kann für komplexe Bedingungen mit anderen Bitarrays kombiniert werden

Beispiel 1

- Darstellung von 20 Werten
- Bitmap
  - Speicherverbrauch (pro Tupel) — 20 Bit
  - Vergleich mit Konstanter — 1 Bitarray lesen
- Bitmapped zur Basis <4,4>
  - Speicherverbrauch — 8 Bit
  - Vergleich mit Konstanter — 2 Bitarrays
- Bitmapped zur Basis <2,4,3>
  - Speicherverbrauch — 9 Bit
  - Vergleich mit Konstanter — 3 Bitarrays
- Bitmapped zur Basis <2,2,2,2,2>
  - Speicherverbrauch — 10 Bit
  - Vergleich mit Konstanter — 5 Bitarrays
- Binärdarstellung
  - Speicherverbrauch — 5 Bit
  - Vergleich mit Konstanter — 5 „Bitarrays“

Join-Index (≠ Index-Join)

- Indexierung von Spalten einer Tabelle A mit Werten einer Tabelle B

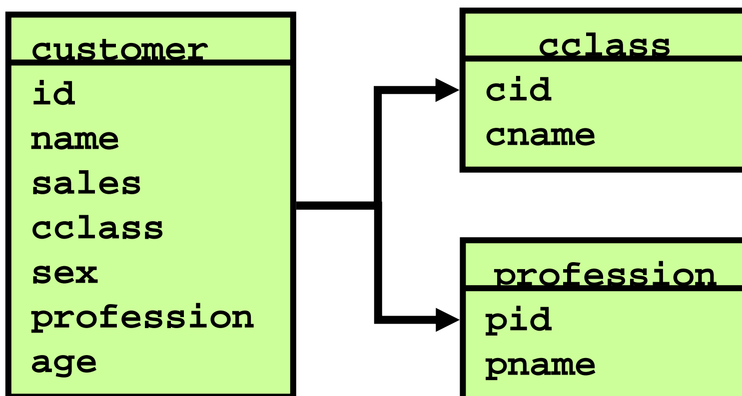



```
CREATE INDEX myIndex ON sales (region.region_name)
USING sales.region_id = region.id
```

(Redbrick)

Bitmapped Join Index

- Unsere bisherigen Beispiele für Bitmap-Indexe decken manche Probleme nicht ab
  - Bedingungen nicht an FK, sondern an „semantische“ Attribute der Dimensionstabellen gerichtet
  - Bitmaps auf FK verhindern dann nicht den Join zu Dimensionstabellen
- Lösung: Bitmapped Join Index



- Bitmaps erstellen für Tupel in `customer` für Werte in `cclass.cname` und in `profession.pname`

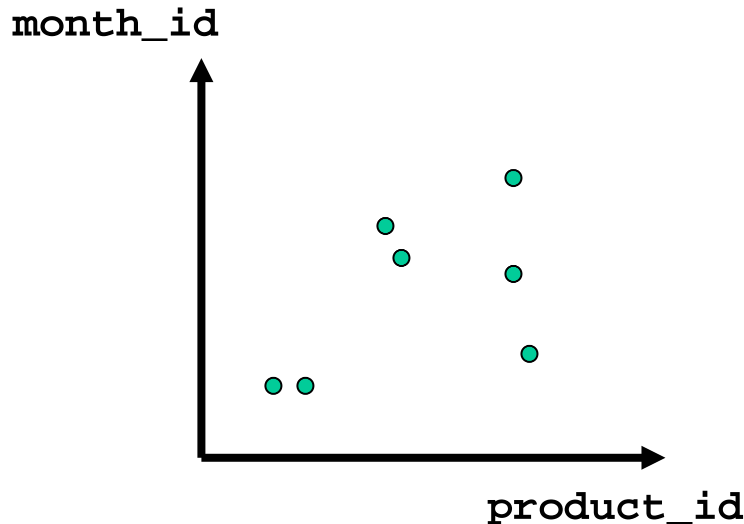
Content of this Lecture

- Introduction to Multidimensional Indexing
- Grid-Files
- Kdb-trees
- Other

Multidimensional Indexing

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

Composite Indexes

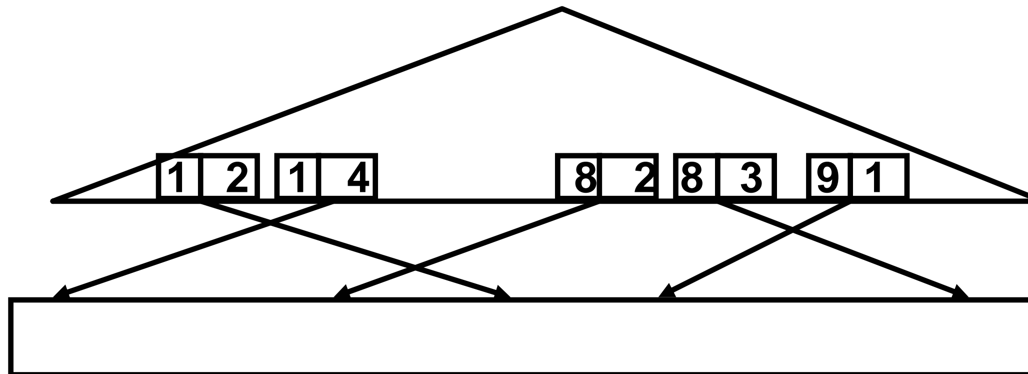


Point	X	Y
P1	2	2
P2	2	2
P3	5	7
P4	5	6
P5	8	6
P6	8	9
P7	9	3

- Imagine composite index on (X, Y)
- Box queries: efficiently supported
- Partial queries
  - All points/rectangles with X coordinate between …
    - Efficiently supported
  - All points/rectangles with Y coordinate between …
    - Not efficiently supported

Composite Index

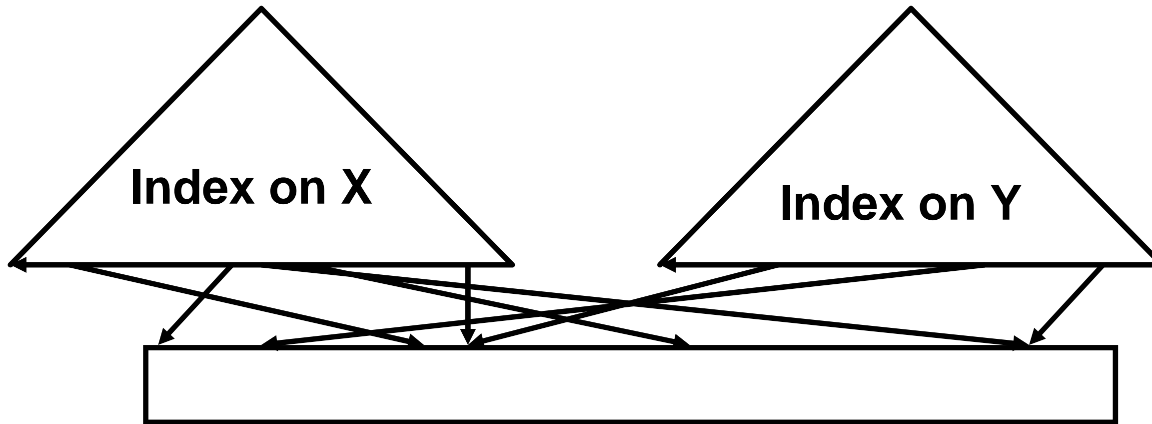
- One index with two attributes (X, Y)



- General
  - Prefix of attribute list in index must be present in query
  - The longer the prefix in the query, the better
- Alternatives
  - Also index (Y, X)
    - 5 attributes: 5! orders
    - Combinatorial explosion, infeasible for more than 2 attributes
  - Use independent indexes on each attribute

Independent Indexes

- One index per attribute



- Partial match query on one attribute: supported
- Partial match query on many attributes
  - Compute TID lists for each attribute
  - Intersect

Independent versus Composite Index

- Consider 3 dimensions of range 1,…,100
  - 1.000.000 points, uniformly (and randomly) distributed
  - Index blocks hold 50 keys or records
  - Index on each attribute has height 4
- Find points with 40≤x≤50, 40≤y≤50, 40≤z≤50
  - Using x-index, we generate TID list |X|~100.000
  - Using y-index, we generate TID list |Y|~100.000
  - Using z-index, we generate TID list |Z|~100.000
  - For each index, we have 4+100.000/50=2004 IO
    - TIDs are sorted in sequential blocks, each holding 50 TIDs
  - Hopefully, we can keep the three lists in main memory
  - Intersection yields app. 1.000 points with 6012 IO
    - Why 1000 points?
- 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 (104)

Conclusion 1

- We want composite indexes
  - Much less IO
  - Things get worse for bigger d
    - TID lists don't fit into main memory – paging, more IO
    - Reading large TIDs lists again and again is more work than scanning relation once
    - Linear scanning of relation might be faster
  - Advantage of composite indexes grows "exponentially" with number of dimensions and selectivity of predicates
  - Things get complicated if data is not uniformly distributed
    - Dependent attributes (age – weight, income, height, …)
    - Clustering of points
    - Histograms

Conclusion 2

- But: For partial queries, we need to index all combinations
  - Impossible
- Solution: Use multidimensional indexes
  - General: Improvement, but no solution
  - “Curse of dimensionality” still valid
    - Most md indexes degrade for many dimensions
    - All md indexes have some worst-case data distribution
      - Usually far from normal or equal distribution
      - Bad space usage, excessive management cost, …
- Alternative: Bitmap indexes
  - Need to load values for all tuples and all dimensions
  - But: Very small memory footprint, fast BIT operations

Multidimensional Indexes

- All dimensions are equally important
- Neighbors in space are (hopefully) stored on nearby blocks
  - Locality property
  - Difficult to achieve, very important
- Supported queries
  - Exact match point queries
  - Partial match point queries
  - Box queries (range queries)
  - Nearest neighbor queries
    - In multidimensional space
- This is different from B-trees
  - All B-trees need a total order on the keys
  - For more than one dimension, no such order exists

Content of this Lecture

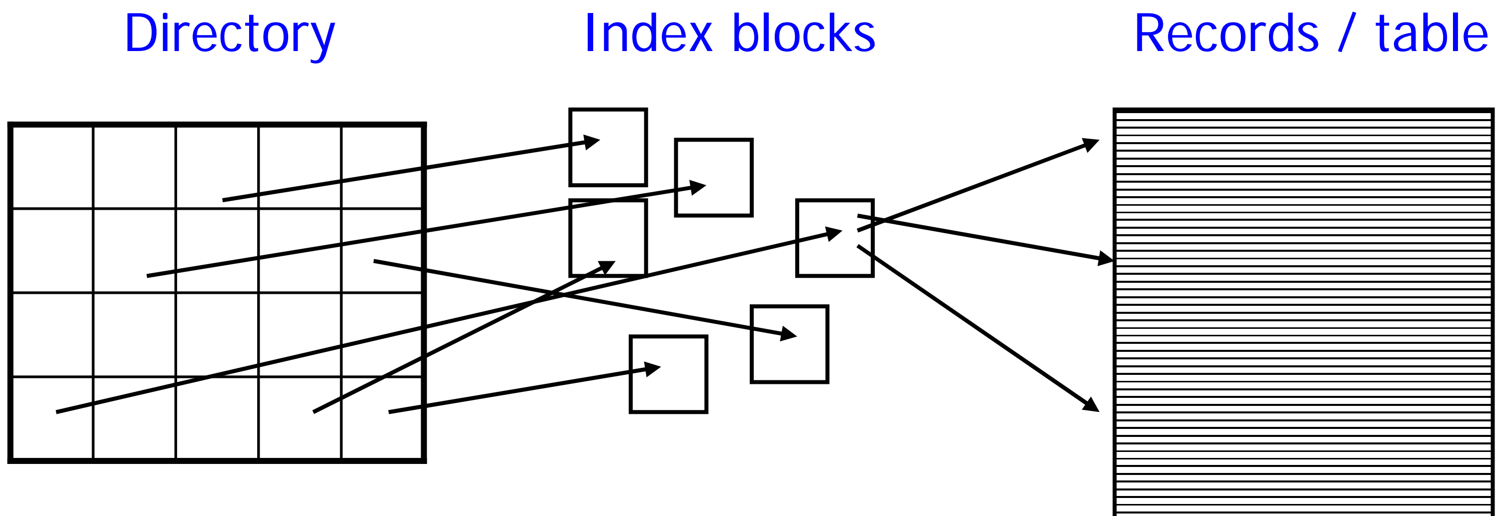
- Introduction to Multidimensional Indexing
- Grid-Files
- Kdb-trees
- Other

Grid-File

- Classical multidimensional index structure
  - Simple: searching, indexing, deleting
  - Good for uniformly distributed data
  - Cannot handle skewed data well
  - Many variations (we will point to different options)
- Design goals
  - Index point objects
  - Support exact, partial match, and neighbor queries
  - Guarantee “two IO” access
    - Under some assumptions
  - Do not prefer any dimension
  - [Adapt dynamically to the number of points]

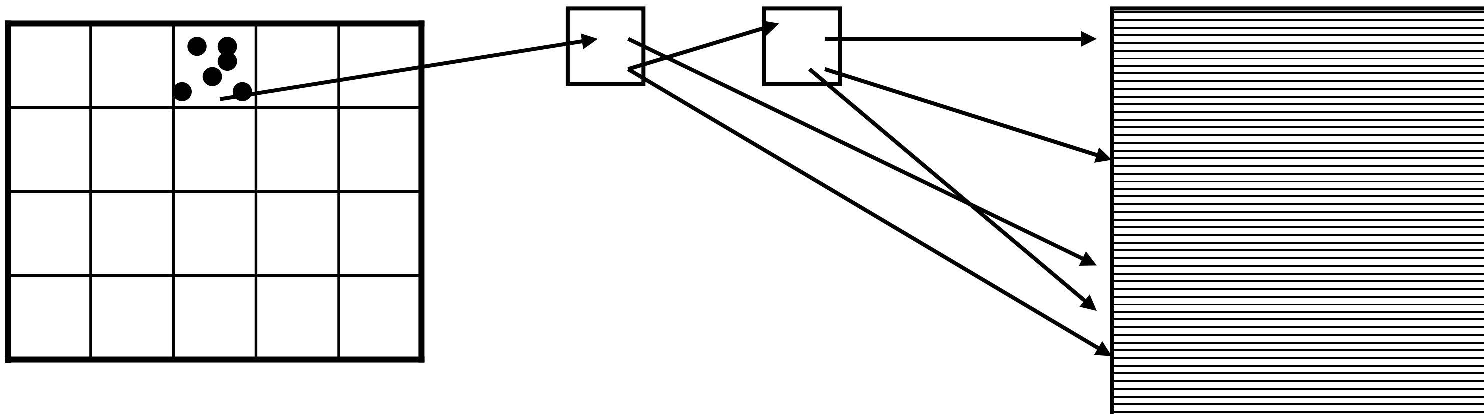
Fixed Grid-File

- [Does not adapt to data distribution at all]
- Idea
  - Split space into equal-spaced cuboids or cells
    - We need maximal and minimal values in all dimensions
  - Directory stores one pointer for each cell
    - Points to block storing pointers to records



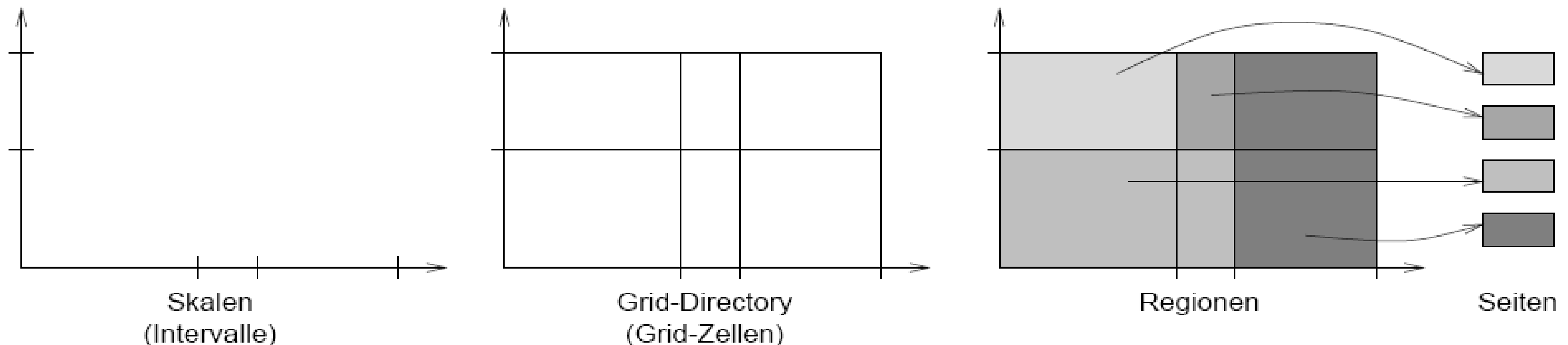
Operations

- Problem: each index block can hold only m pointers
- Deleting a point
  - Compute cell using coordinates
  - Search cell in directory and load index block
  - Search point and delete, if present
- Inserting a point
  - Find cell and load index block
  - If free space: insert pointer
  - If no free space: generate overflow block
    - No adaptation to skewed data distributions
    - May degenerate to linear search



Principle of Grid-Files

- Partition each dimension into disjoint intervals (scales)
- The intersection of all intervals defines all grid cells
  - Convex d-dimensional cuboids
  - Grid directory holds pointer to index blocks per cell
  - When cell overflows – split cell (no overflow blocks)
  - Each point falls into exactly one grid cell
  - Many grid cells (a region) may point to the same index block



Exact Point Search

- Compute grid cell coordinate
  - We keep scales for each dimension in memory
  - Looking up point coordinate in scales gives coordinates for each dimension
  - Map coordinate to index block address on disk
    - We assume that the directory is in main memory
    - Otherwise: Use B-tree with coordinates as keys and pointer to index block as value
- Load index block (1st IO)
  - Search point in index block
- Access record following pointer (2nd IO)

Range Query, Partial Match Query

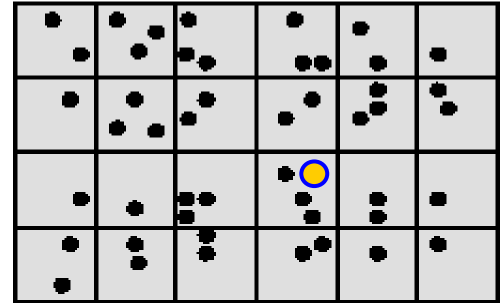
- Range query
  - Compute grid cell coordinate for each end point
  - All grid directory entries in that range may contain qualifying points
- Partial match query
  - Compute partial grid cell coordinates
  - All grid directory entries with these coordinates may contain points

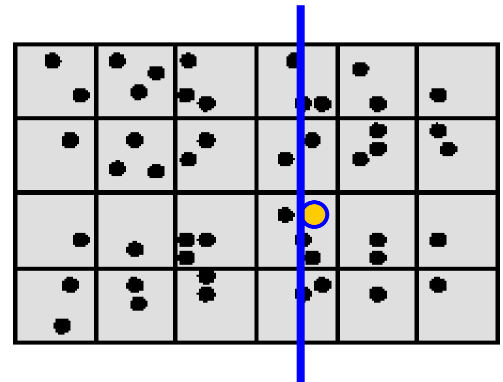
Inserting Points

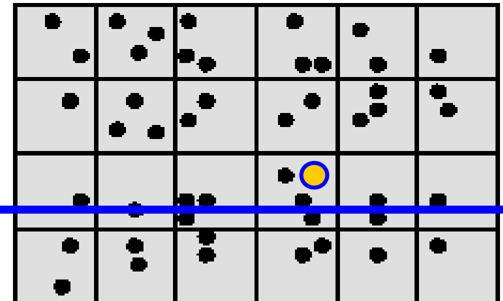
- If index block has space – no problem
- Otherwise (1st split): Split cuboid
  - Choose a dimension and a coordinate (scale) to split
  - Create new index block and distribute points according to the chosen split
  - Insert point
  - Keep in memory new slits as new scale
  - Implicitly split all other affected grid cells
- Choice of "good" dimension and coordinate is very difficult
  - Optimally, we would like to split as many very full index blocks as evenly as possible
  - This is an optimization problem in itself

Example

- Imagine one index block holds 3 pointers
  - Usually we have unevenly spaced intervals
- New point causes overflow
- Where should we split?
- 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 $O(d_i^{n-1})$ regions





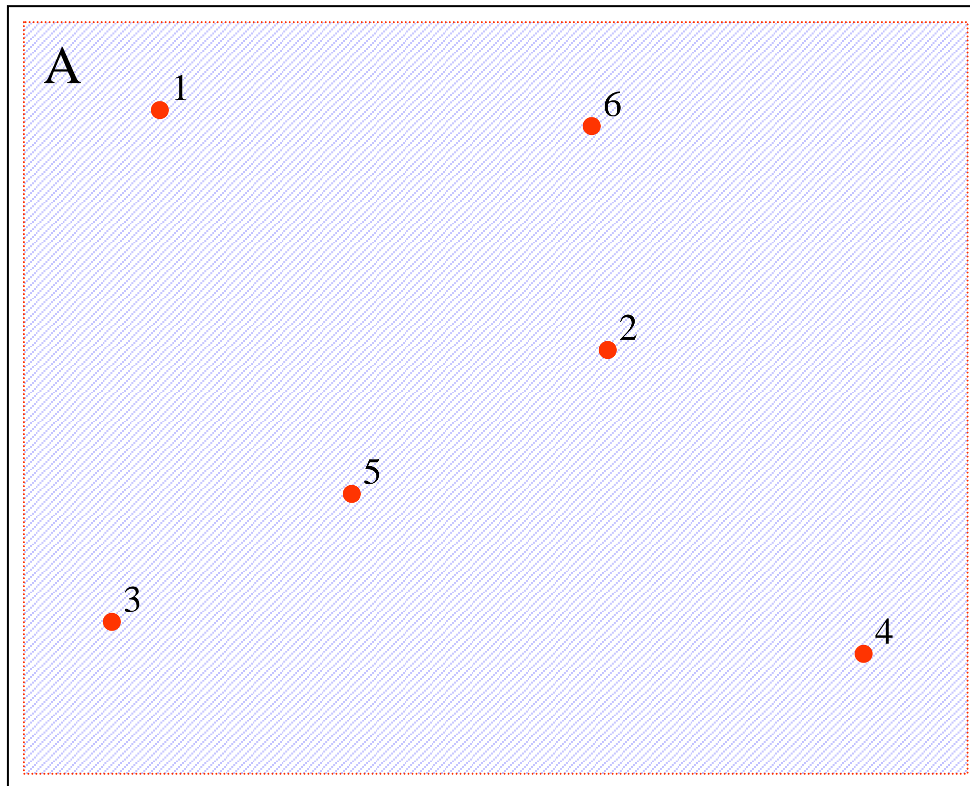


Inserting Points -2-

- Otherwise (none 1st-split)
  - Check borders of index block wrt to scales
  - Borders form a minimal cuboid?
    - Proceed as if it was a 1st-time split
  - Otherwise
    - Split region of index block into smaller cells
    - Consider only not yet realized scales as potential split
    - Create new index block, redistribute points

Grid-File Example 1 (from Johannes Gehrke)

(N=6)

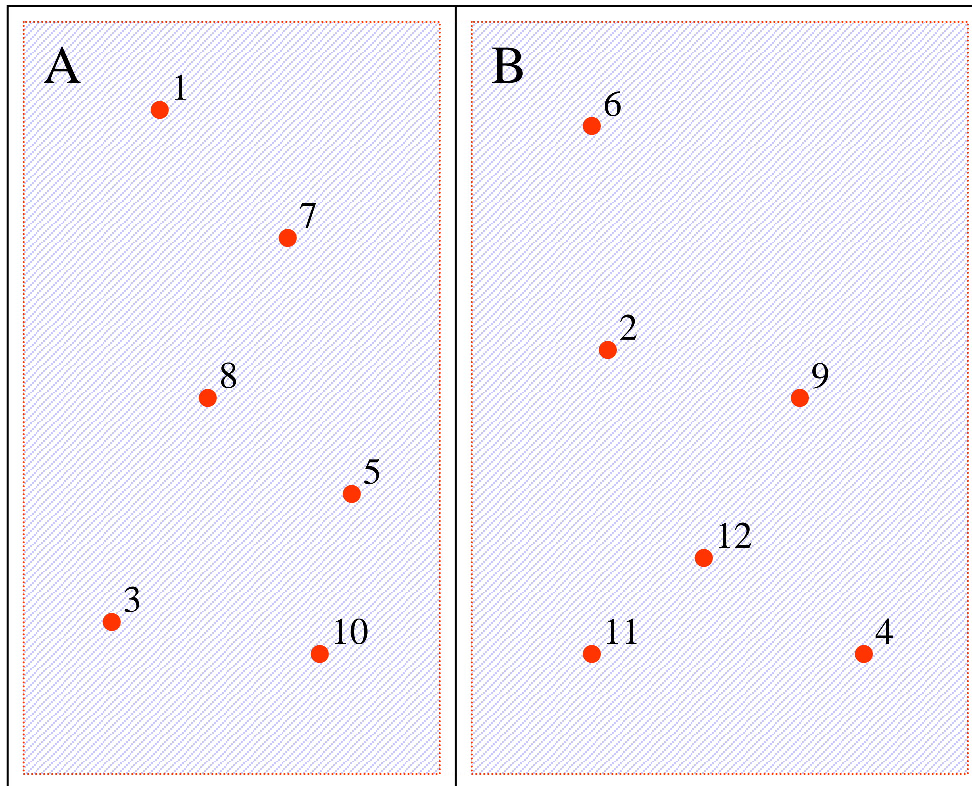


A

A	1	2	3	4	5	6

Grid-File Example 2

(N=6)

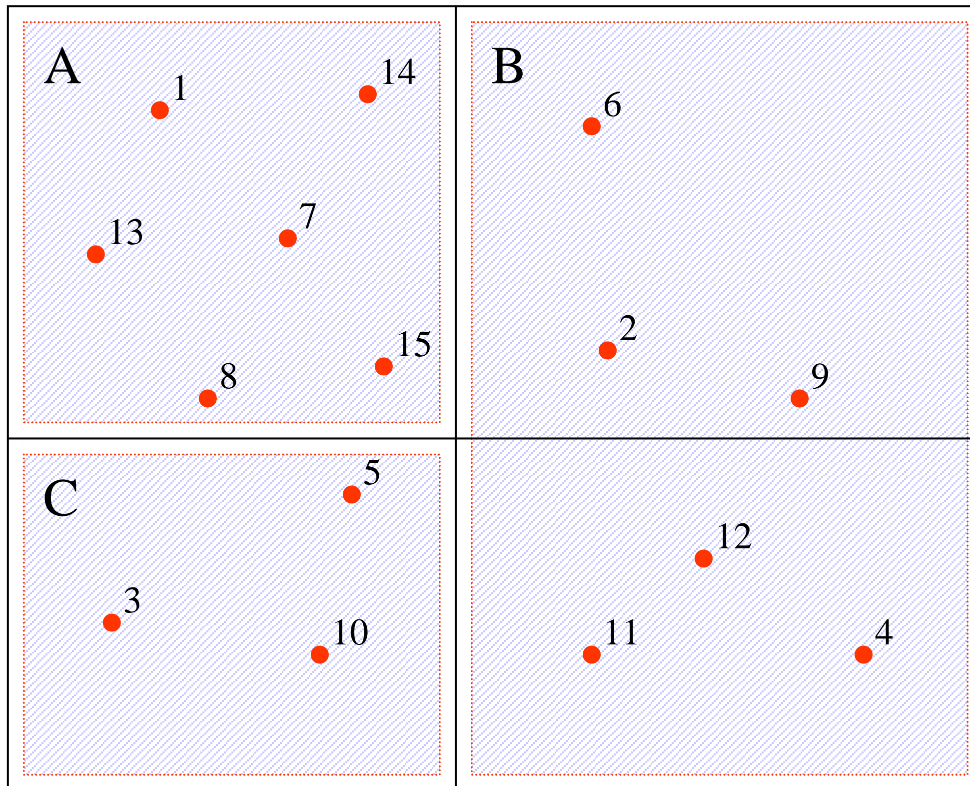


A	B

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

Grid-File Example 3

(N=6)

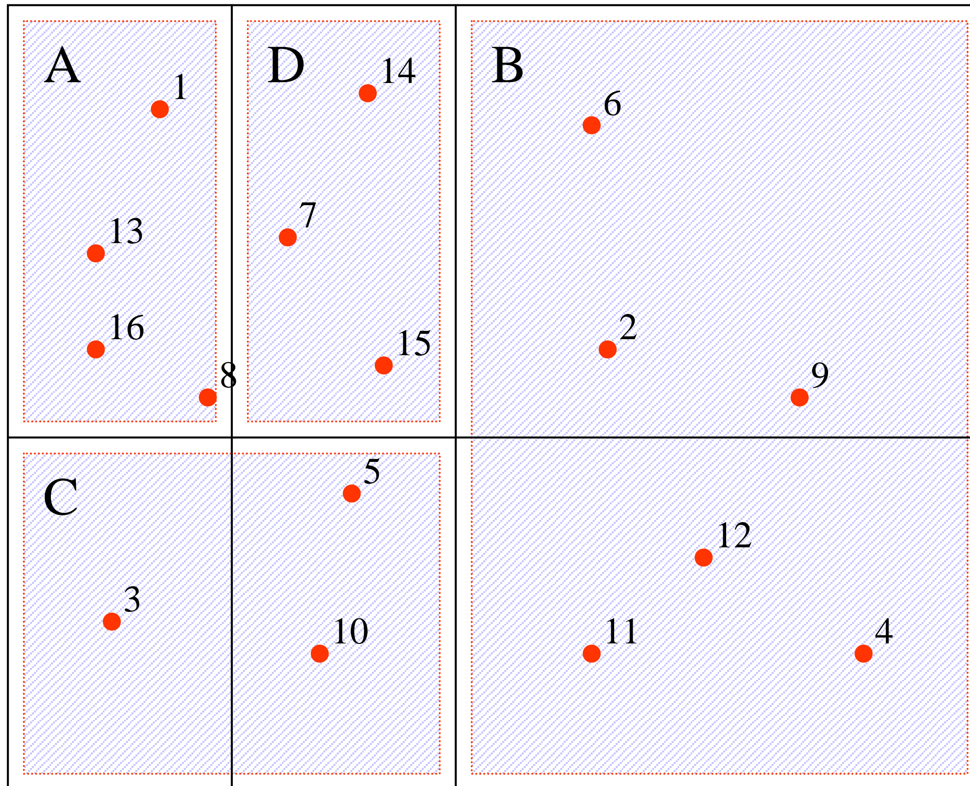


A	B
C	B

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

Grid-File Example 4

(N=6)

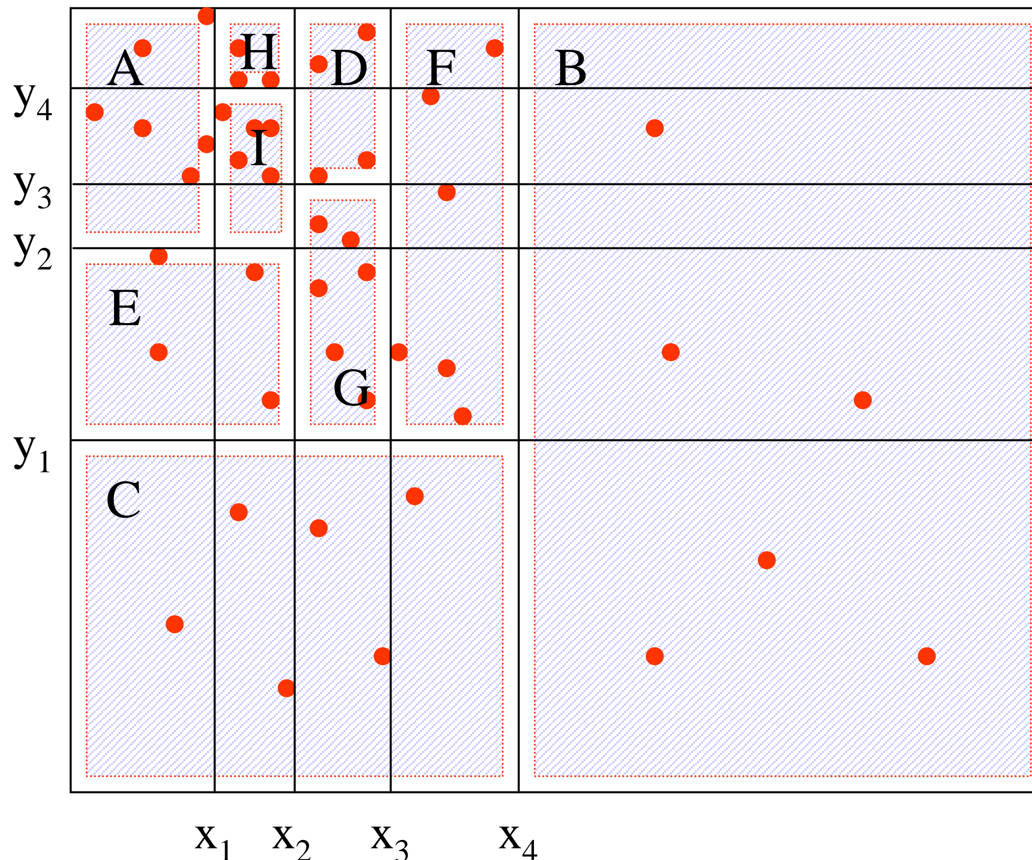


A	D	B
C	C	B

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

Deleting Points

- Search point and delete
- If regions become “almost” empty, merge index blocks
  - A merge is the removal of a split
  - Must build larger convex regions
  - This can become very difficult
    - Potentially, more than two regions need to be merged to keep convexity condition
  - Example:
    Where can we merge regions??

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

Conclusions

- Grid-Files always split at hyperplanes parallel to the dimension axes
  - This is not always optimal
  - Use others than rectangles as cells: circles, polygons, etc.
    - But forms might not disjointly fill the space any more
    - Allow overlaps - R trees
- Good: Good bucket fill degrees
- Bad: Grid directory grows very fast
- Two IO guarantees only holds when directory fits into memory
- Each split finally becomes valid for the entire grid
  - Need not be realized immediately, but restricts later choices
  - Bad adaptation to “unevenly skewed” data
  - The more dimensions, the worse

Content of this Lecture

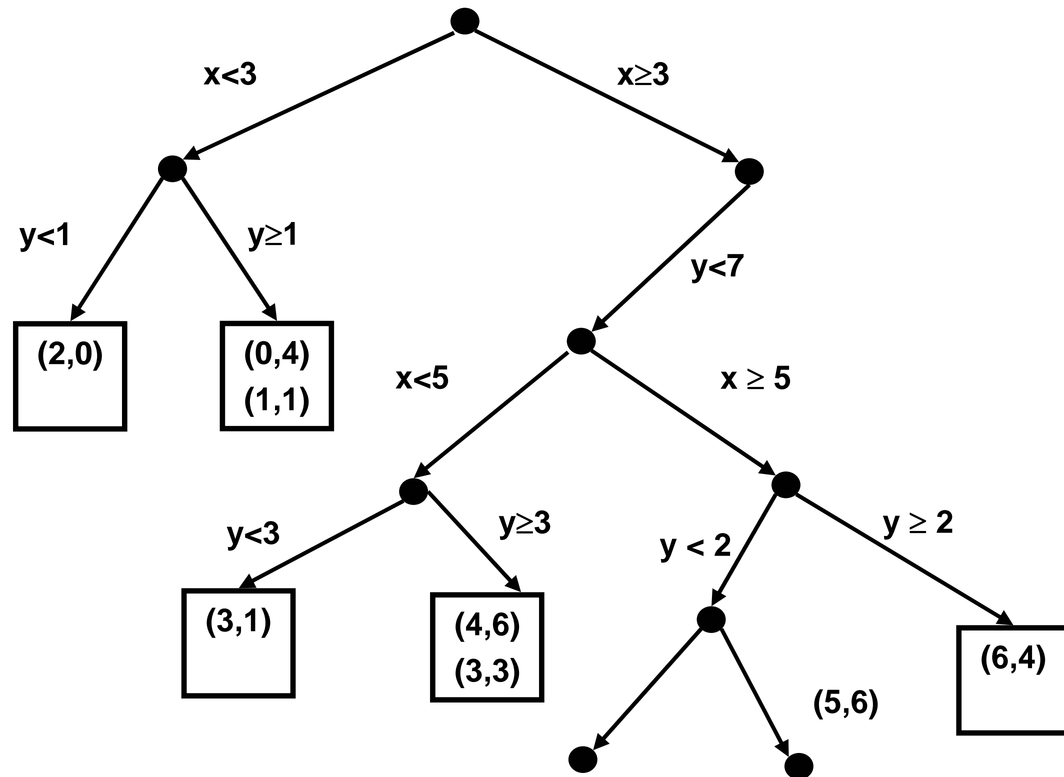
- Introduction to multidimensional indexing
- Grid-Files
- Kdb-trees
- Other

kd-tree

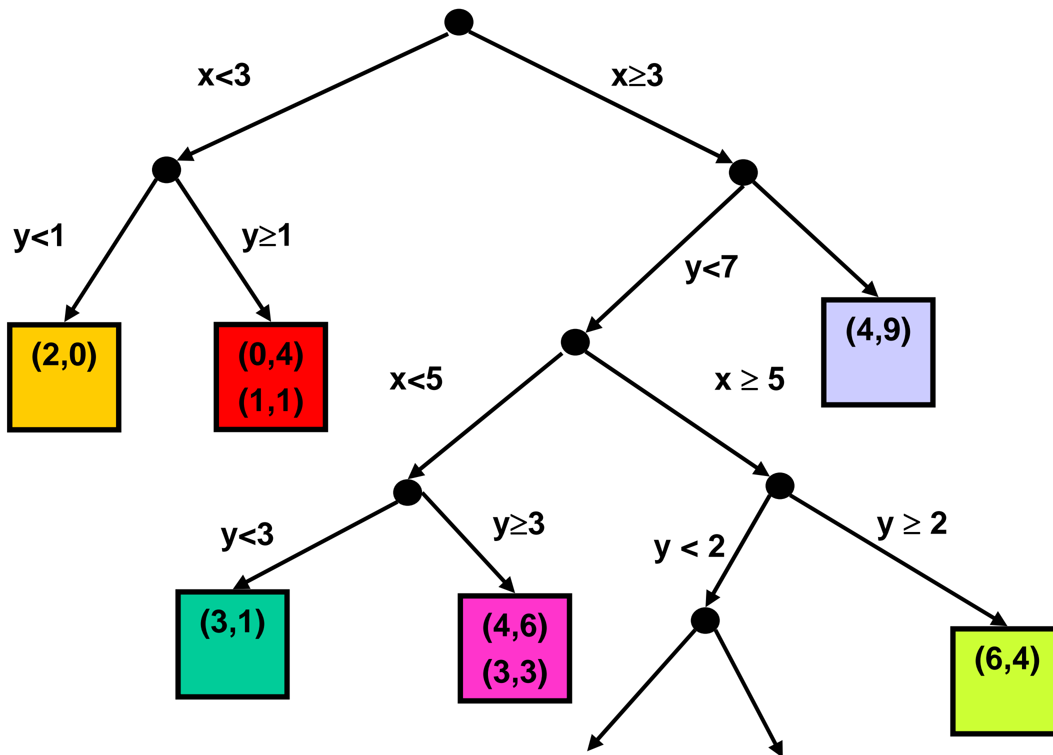
- Grid-File disadvantages
  - All hyperregions of the n-dimensional space are eventually split at the same dimension/position
    - Although not all regions are actually performing the split
  - First cell that overflows determines split
  - This choice is global and never undone
- kd-trees
  - Multidimensional variation of binary search trees
  - Hierarchical splitting of space into regions
  - Regions in different subtrees may use different split positions
  - Better adaptation to clustering of data than Grid-Files
  - kd-tree is a main memory data structure
    - kdb-trees: IO-optimization for layout of inner nodes (later)

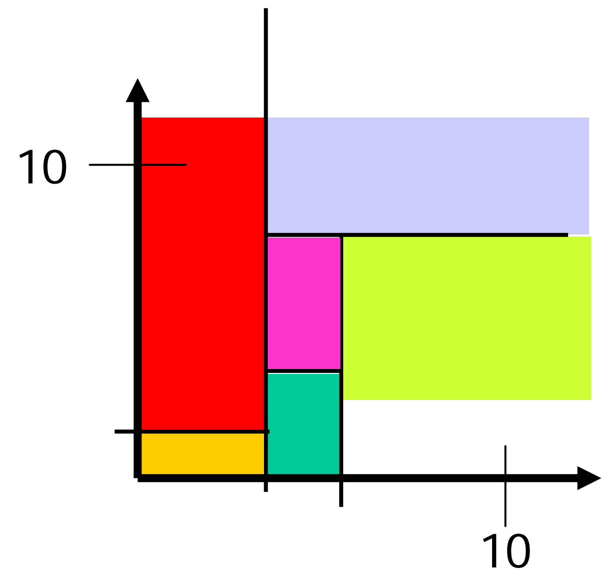
kd-tree: General Idea

- Binary, rooted tree
- Each inner node has two children
- Path is selected based on a pair (dimension / value)
- Dimensions need not be statically assigned to levels of the tree
  - Can be rotating, random …
- Data points are only stored in leaves
- Each leave stores points in a n-dimensional hypercube with m border planes (m≤n)

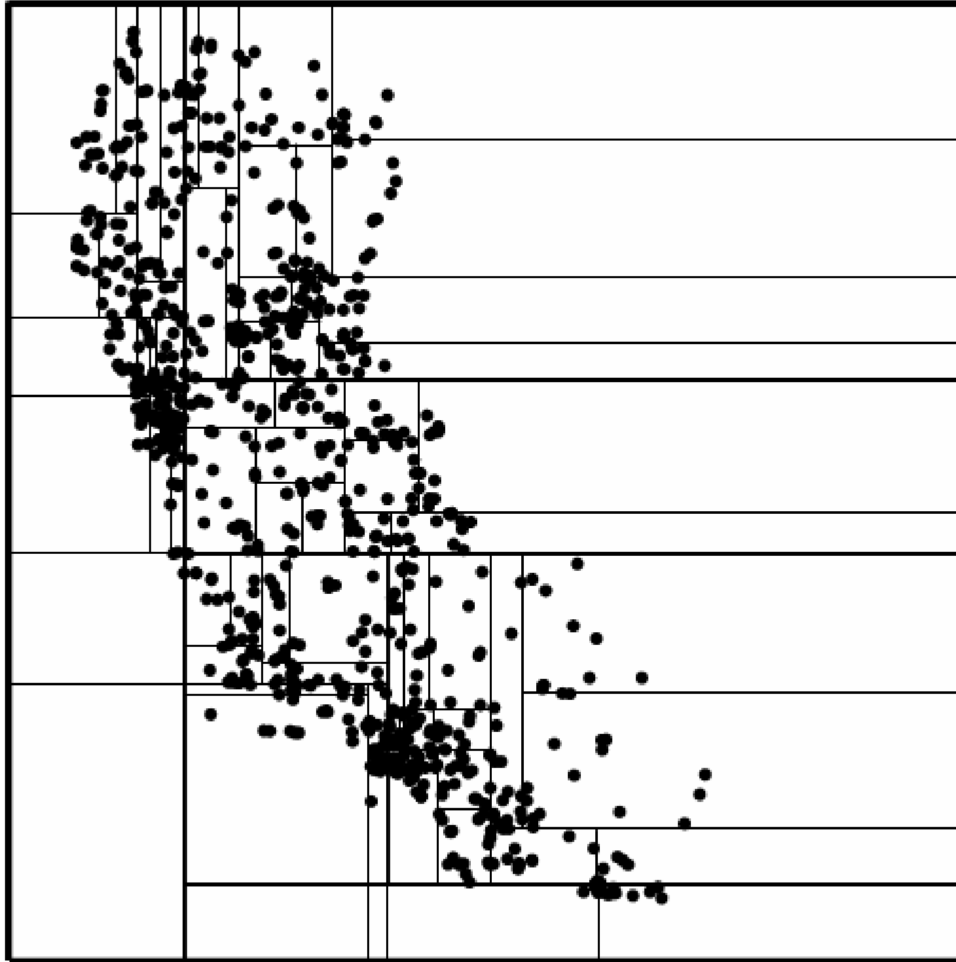


Example – the Brick wall





Local Adaptation



kd-tree Search Operations

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

kd-tree Search Operations

- Exact point search
  - In each inner node, decide direction based on split condition
  - Search leaf for searched point
- 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
    - Leads to multiple search paths
- Range query
  - Follow all children matching the range conditions
    - Again: multiple search paths
- Nearest Neighborhood
  - Chose "close-enough" range and perform range query
  - If no success, iteratively broaden range

kd-tree Insertion

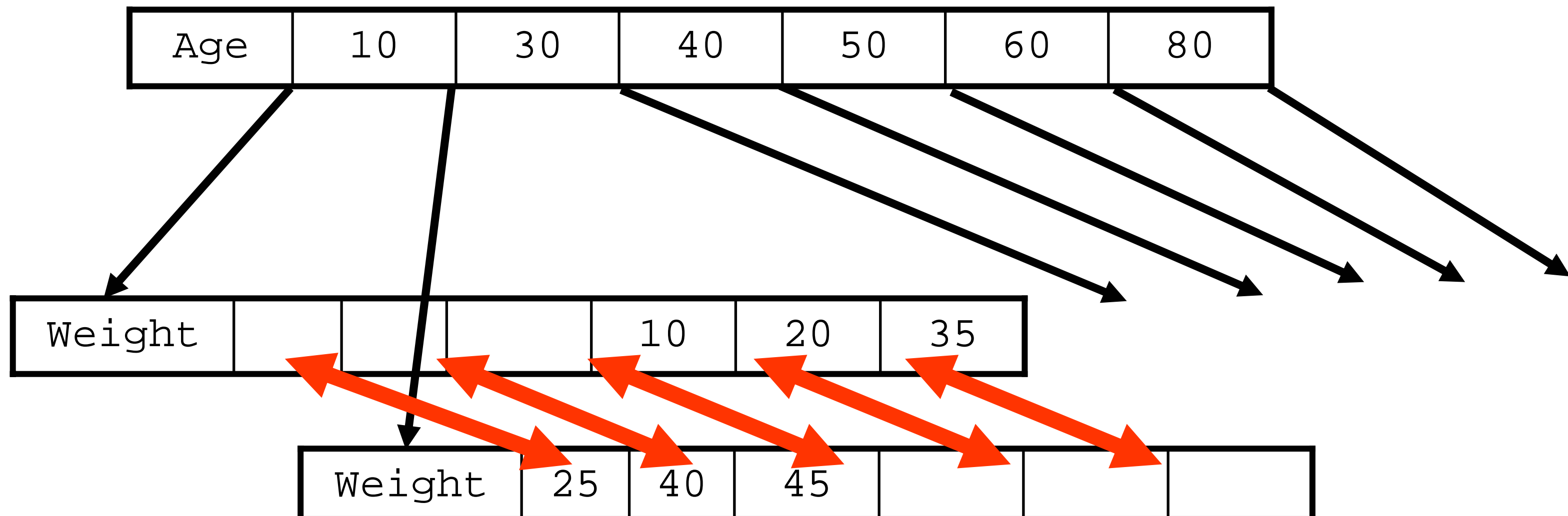
- Inserting a point
  - Search data block of leaf
  - If space available – done
  - Otherwise, chose split dimension and position for this block
    - This is a local decision, but remains stable for the future of the subtree
    - Find dimension and split that divides set of points into two sets
      - Consider current points and split in two 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, religion, place of birth, …
    - But we don't know which points will be inserted in future

Persistent kd-Trees

- Managing a kd-Tree on disks
- Leave nodes are disk blocks
  - Fine
- Inner nodes
  - We should not store just the split info in an entire block
  - So what?

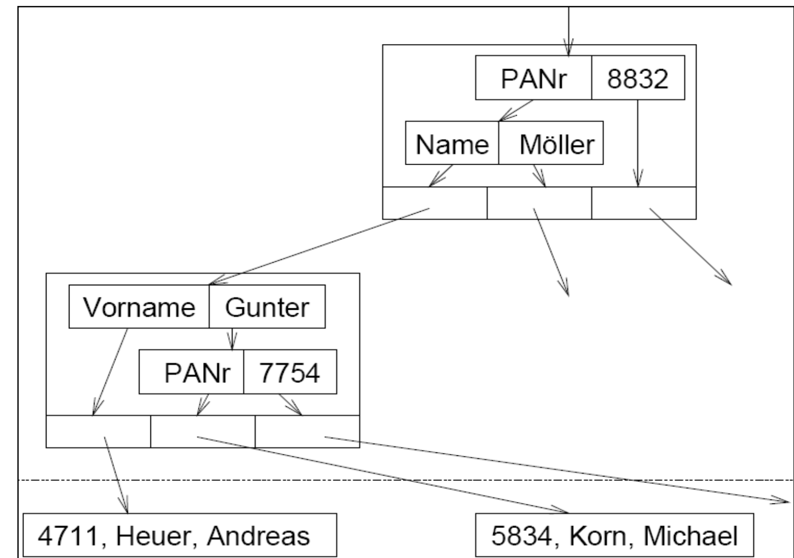
Fill Inner Blocks

- Option 1: Multiway branching
  - Split chosen dimension at r positions
    - r: Number of pointer/value pairs fitting in block
    - BUT: When sibling nodes need to be merged,
      - Split points of children usually are incompatible
      - Reorganization of subtrees required



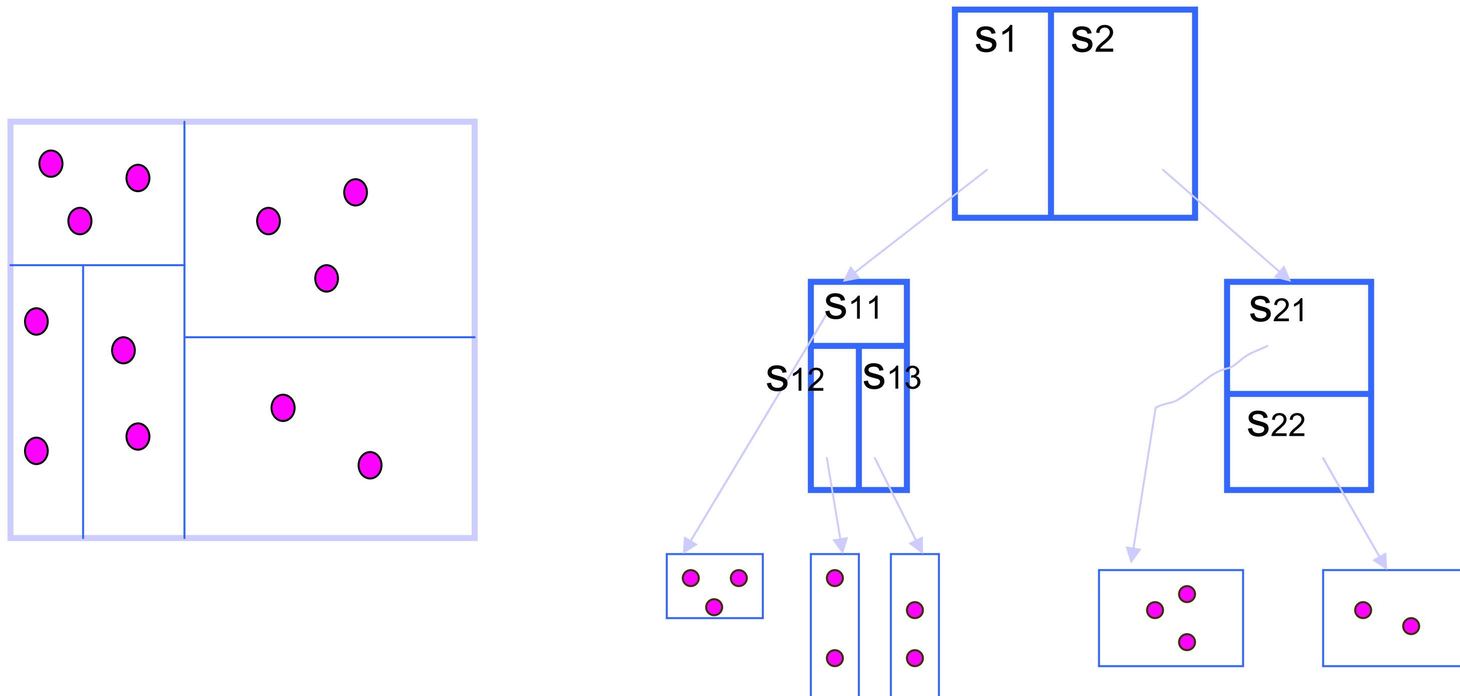
kdb-trees

- Option 2: Store entire subtrees in one block
  - Inner nodes still have only two children
    - But those are (usually) stored in the same block
    - We need to “map” nodes to trees
  - kdb-tree: inner nodes store kd-trees
- Operations
  - Searching: As with kd-trees
    - But better IO complexity
  - Insertion/Deletion
    - Complex scheme for keeping tree balanced



Another View

- Inner nodes define (possibly open) bounding boxes on subtrees
- Kdb-tree is a hierarchical index structure



Conclusion

- kdb-trees can be perfectly 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 unnecessary large
  - No guarantee for lowest fill degree as in b* tree

Content of this Lecture

- Introduction to multidimensional indexing
- Grid-Files
- Kdb-trees
- Other

Partitioned Hashing

- Partitioned Hashing
  - Let A_1 , A_2 ,…, A_k be search keys
  - Define a hash function for each A_i ; interpret result as bit string
  - Global hash key: concatenation of the attribute bit strings
  - Definition
    - Let $h(A_i)$ map each A_i into a integer with b_i bit
    - Let $b=\sum b_i$ (length of global hash key in bits)
    - The global hash function

      $$h(v_1, v_2, \ldots, v_k) \rightarrow [0, \ldots, 2^b-1]$$

      is defined as

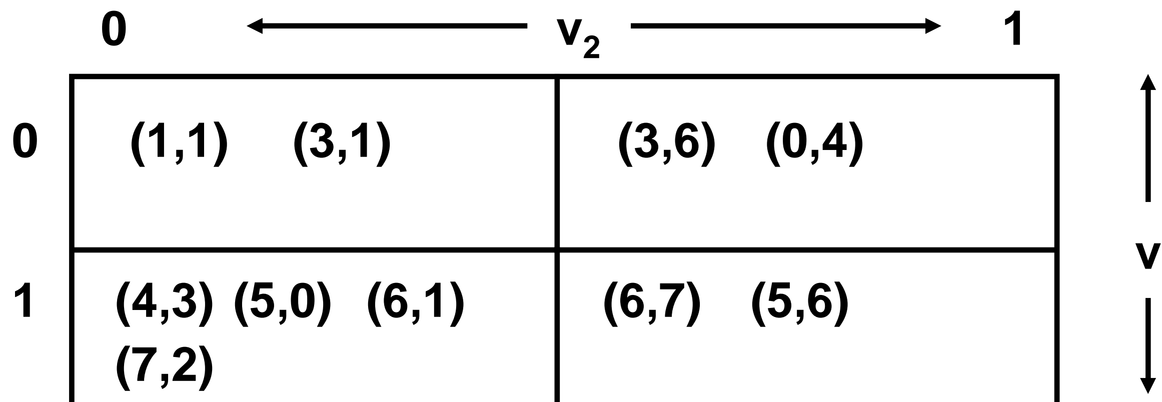
      $$h(v_1, v_2, \ldots, v_k) = h_1(v_1) \oplus h_2(v_2) \oplus \ldots \oplus h_k(v_k)$$
    - We need $B = 2^b$ buckets
      - Static address space – dynamic structures later

Example

- We want to store points
  - (3,6), (6,7), (1,1), (3,1), (5,6), (4,3), (5,0), (6,1), (0,4), (7,2)
- Let hash function h_1, h_2 be

$$h_i(v_i) = \begin{cases} 0 & \text{if } 0 \le v_i \le 3 \\ 1 & \text{otherwise} \end{cases}$$

- Thus, there are 4 buckets with address 00, 01, 10, 11



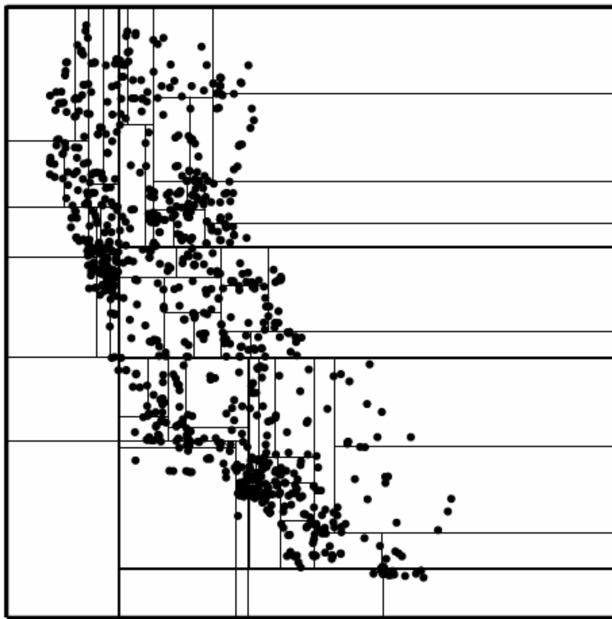
Queries with Partitioned Hashing

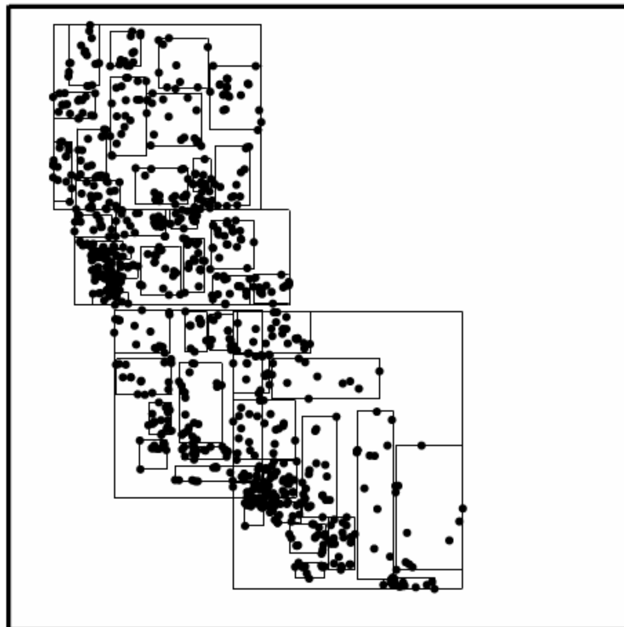
- Exact point queries
  - Direct access to bucket possible
- Partial match queries
  - Only parts of the global hash key are determined
  - Use those as filter; scan all buckets passing the filter
  - Let $c = \sum b_i$ be the number of unspecified bits
    - Then 2^c buckets must be searched
    - These are certainly not ordered (ordered on what?) – random IO
- Range queries
  - Not supported, if hash function doesn't preserve order

Humboldt Universität Informatik

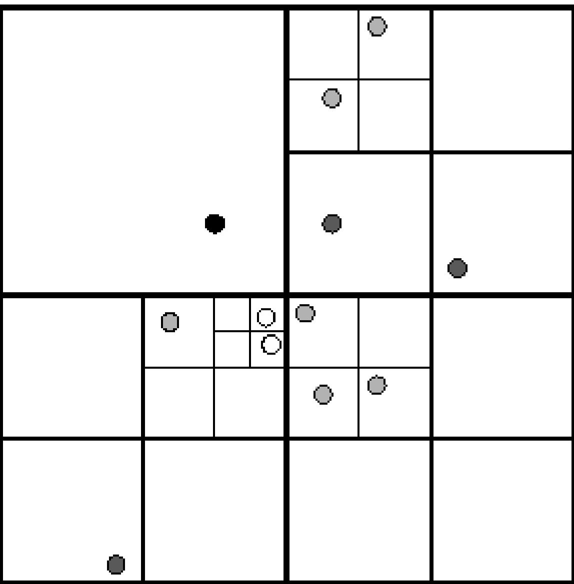
R-trees

- Can store geometric objects (with size) as well as points
- Each object is stored in exactly one region on each level
- Since sized objects may overlap, R tree regions may overlap
- Only those hyperregions containing data objects are represented
- Many variations (see literature)

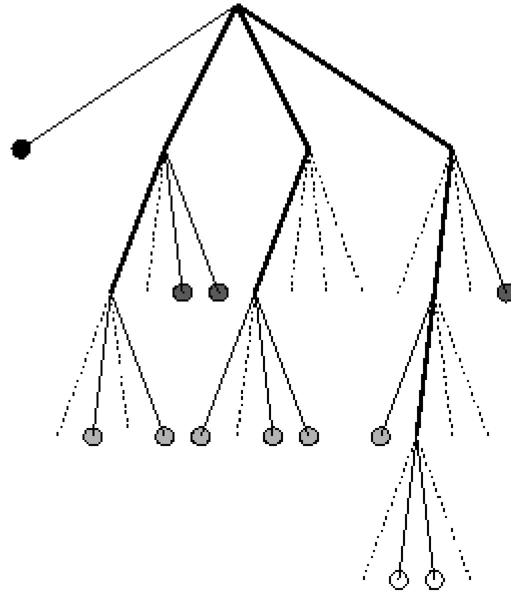




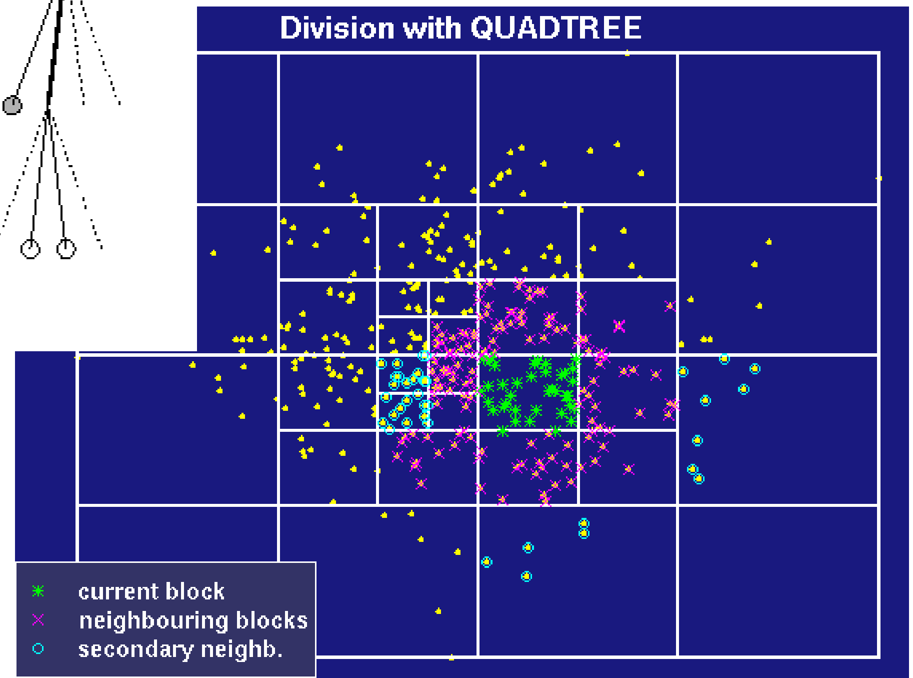
Quad-tree



(a)



(b)



http://www.droxley.freeserve.co.uk/ http://puddle.mit.edu/~glenn/kirill/SAGAHTML/gallery2.html

Multidimensional Data Structures Wrap-Up

- We only scratched the surface
- Other: X tree, hb tree, R+ tree, UB tree, ...
  - Store objects more than once; other than rectangular shapes; map coordinates into integers; ...
- Curse of dimensionality
  - Your intuition plays tricks on you
  - The more dimensions, the more difficult
    - Balancing the tree, finding MBBs, split decisions, etc.
  - All structures begin to degenerate somehow
  - Often, linear scanning of objects is quicker
    - Or: Compute lower-dimensional, relationship-preserving approximations of objects and filter on those