Datenbanksysteme II: Multidimensional Index Structures 1

Ulf Leser

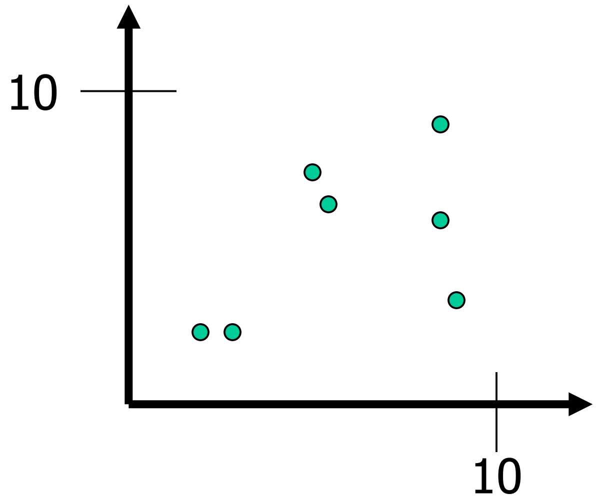
Content of this Lecture

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

Multidimensional Indexing

- Access methods so far support access on attribute(s) for
  - Point query: Attribute = const (Hashing and B+ Tree)
  - Range query: $const_1 \leq$ Attribute $\leq const_2$ (B+ Tree)
- What about more complex queries?
  - Point query on more than one attribute
    - Combined through AND (intersection) or OR (union)
  - Range query on more than one attribute
  - Queries for objects with size
    - "Sale" is a point in a multidimensional space
      - Time, location, product, …
    - Geometric objects have size: rectangle, cubes, polygons, …
  - Similarity queries: Most similar object, closest object, …

Example: 2D Points



Point	X	Y
P1	2	2
P2	2,5	2
P3	4,5	7
P4	4,7	6,5
P5	8	6
P6	8	9
P7	8,3	3

- Objects are points in a 2D space
- Queries
  - Exact: Find all points with coordinates (A1, B1)
  - Box: Find all points in a given rectangle within (A1, B1), (A2, B2)
  - Partial: Find all points with X (Y) coordinate between …

Definitions

- Exact Query: Conjunction of equality condition on every attribute

```
SELECT * FROM POINT
WHERE  a=x and b=y
```

- Range Query: Conjunction of two comparisons on one attribute defining a non-empty interval

```
… WHERE x≥a and x≤b
```

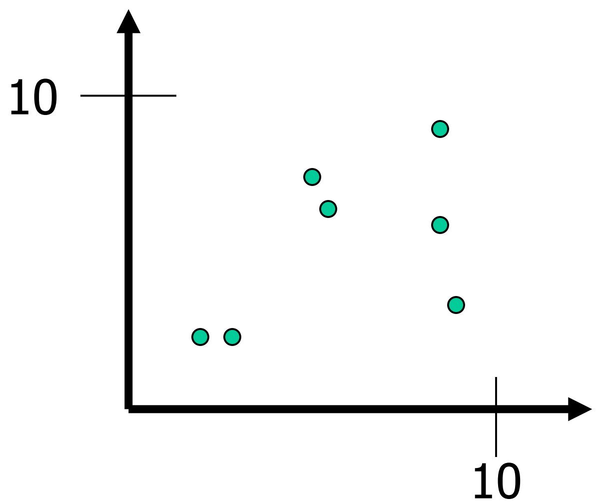
- Box query: Conjunction of range queries in every dimension

```
… a1≤x and b1≤y and
  a2≥x and b2≥y
```

- Partial query: All other

```
… a1≤x and b1≤y
```

Option 1: Composite Index

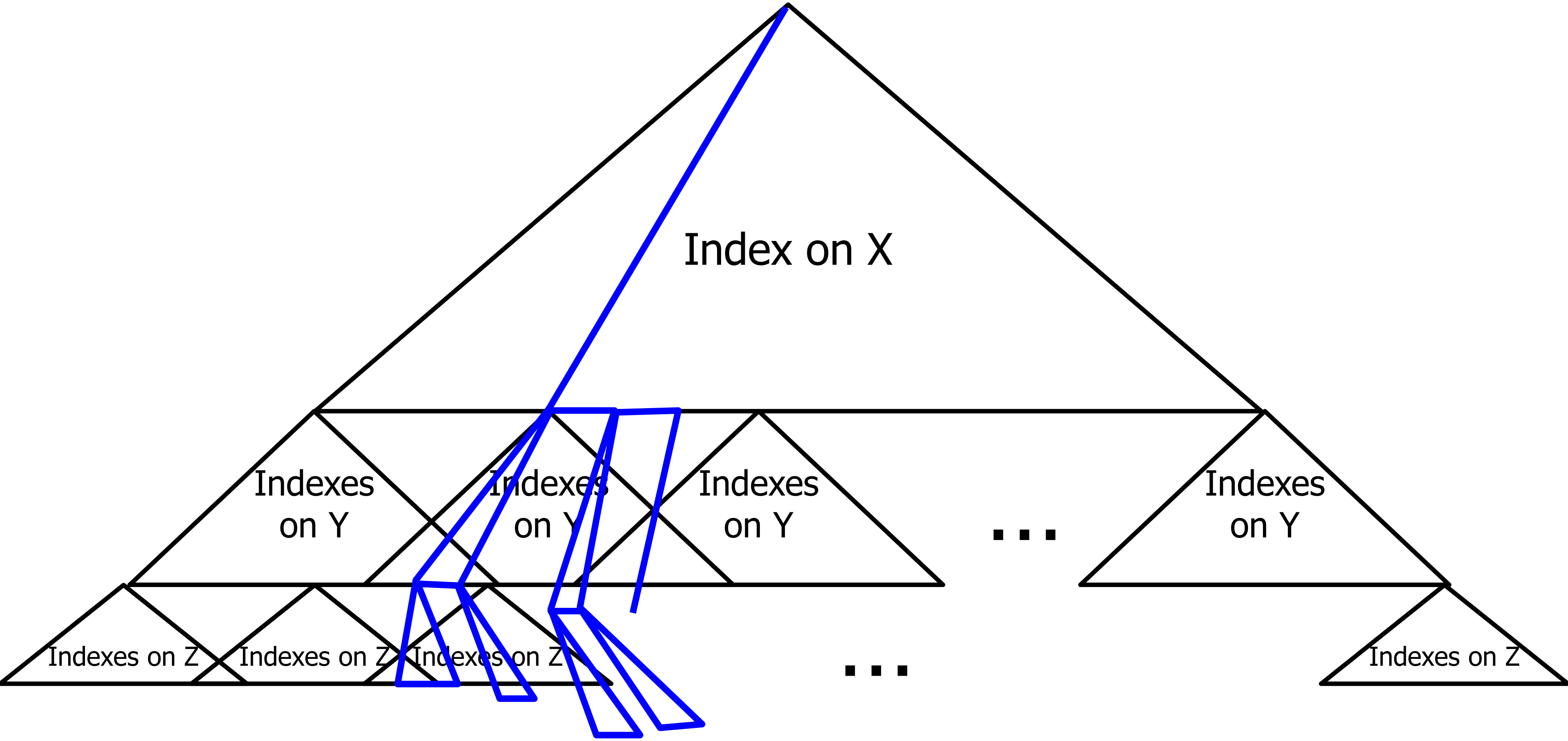



```
CREATE INDEX
ON point(x,y)
```

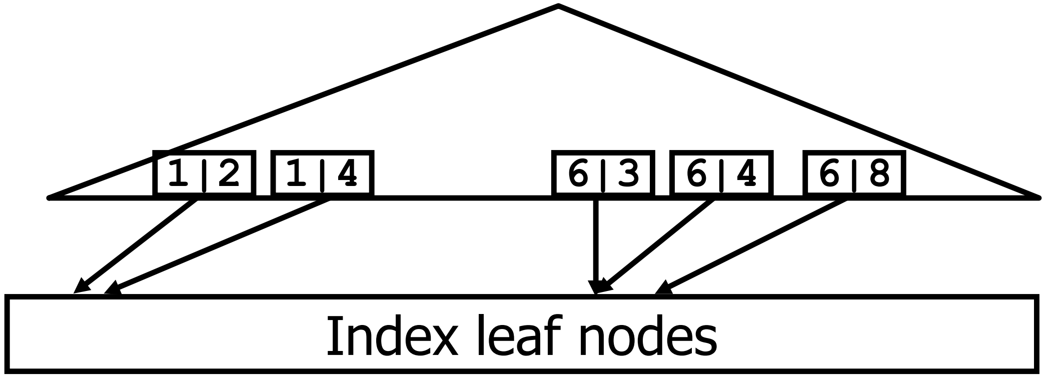
Point	X	Y
P1	2	2
P2	2,5	2
P3	4,5	7
P4	4,7	6,5
P5	8	6
P6	8	9
P7	8,3	3

- Exact queries: Efficiently supported
- Box queries: Efficiently supported
- Partial query
  - All points with X coordinate between …: Efficiently supported
  - All points with Y coordinate between …: Not efficiently supported

Composite Index

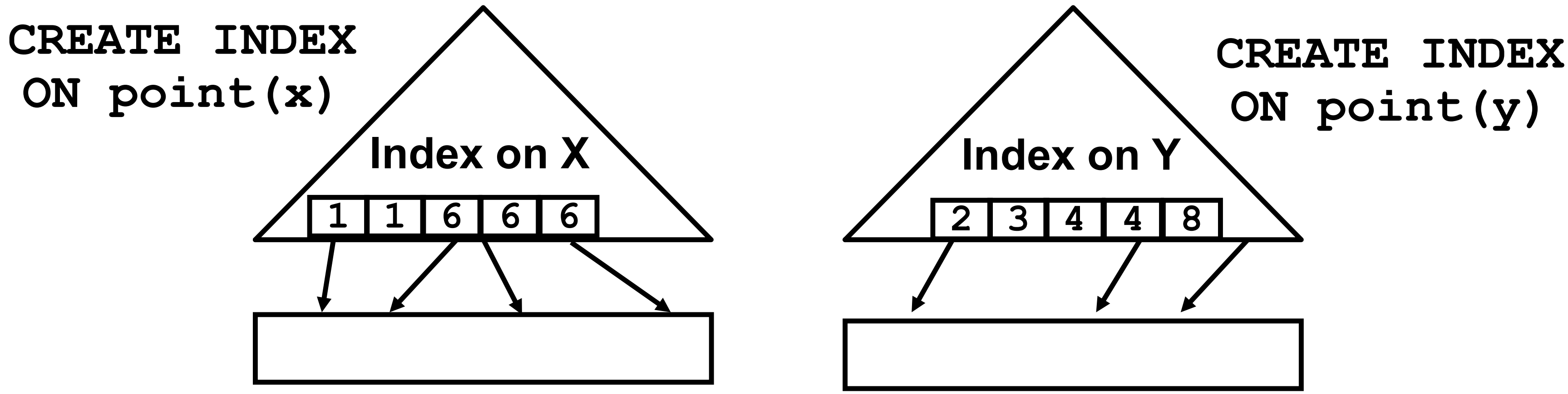


Composite Index



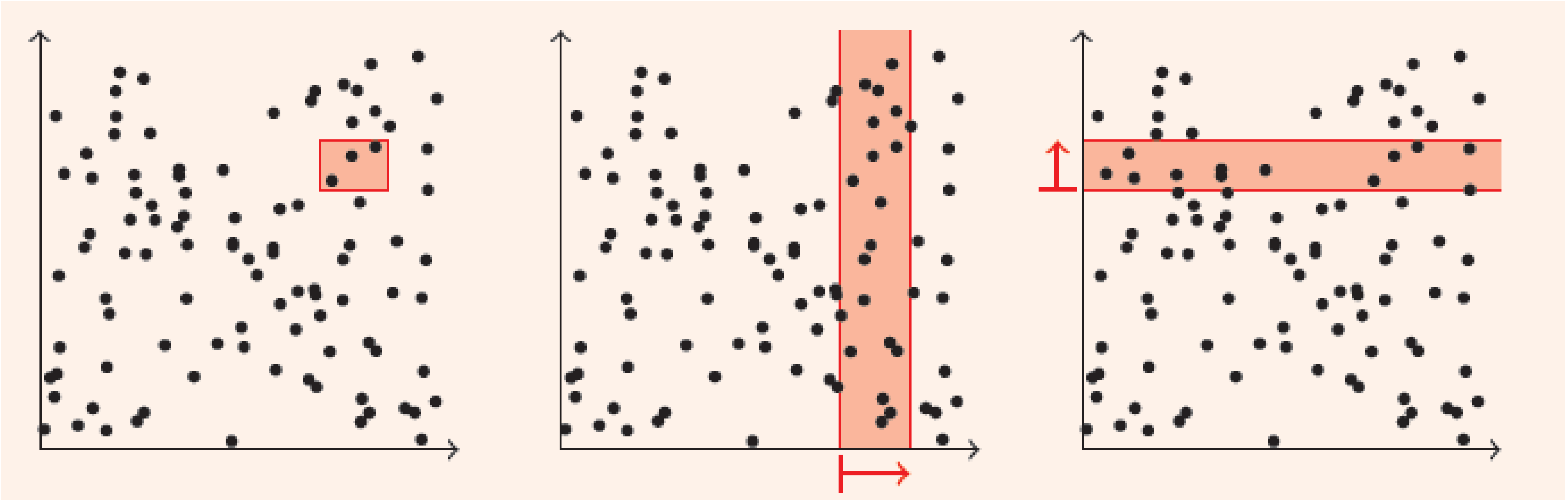
- Usage
  - Prefix of attribute list in index must be present in query
  - The longer the prefix, the more efficient the evaluation
- Alternatives
  - Also build index tab(Y, X) – one for every possible prefix
    - Combinatorial explosion for more than two attributes
  - Use independent indexes on each attribute

Option 2: Independent Indexes



- Exact query: Not efficient
  - Compute TID lists for each attribute
  - Intersect
- Box query: Not efficient (compute ranges, intersect)
- Partial query: Not efficient with more than one dimension

Intuition



Source: T. Grust, 2010

Example – Independent Index

- Data
  - 3 dimensions of range 1,...,100
  - 1.000.000 points, randomly distributed
  - Index leaves holding k=50 keys or records
- Assume three independent indexes
- Box query: Points with $40 \leq x \leq 50$, $40 \leq y \leq 50$, $40 \leq z \leq 50$
  - Each of the three B+-indexes has height 4
  - 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
  - Hopefully, we can keep the three lists in main memory
  - Intersection yields app. 1.000 points, together 6012 IO

Example – Composite index (X,Y,Z)

- Key length increases – assume k=30 (or 10 / more dims)
- Index is higher: Height ~ 5 (6)
  - Worst case – index blocks only 50% filled
- We descend in 5 IO to leaves, read 10 points (1 IO), ascend to Y-axis (2 IO – but cached), descend to leaves (2 IO), read 10 points (1 IO) …
- We do this 10*10 times
- Altogether
  - k=30 => app. 3+100*(2+1) ~ 303 IO
    - Compared to 6012 for independent indexes!
  - k=10 => app. 4+100*(3+1) ~ 404 IO

Conclusion

- We want composite indexes: Less IO
  - Benefit grows for highly selective queries
  - But: If selectivity is low, scanning of relation is faster anyway
    - Sequential versus random IO
- For partial match queries, we would need to index all prefixes – not feasible
- Solution: Use multidimensional index structures (MDIS)

Multidimensional Index Structures

- Specialized IS for MD-objects with or without extend
  - Points versus shapes
  - Should have no priority or preferred dimensions
  - Should adapt to uneven and changing data distribution
  - Should have low worst case complexity (balanced structures)
  - Should not use too much space
  - Locality: Neighbors in space are stored nearby on disk (memory)
    - In an ideal world, we would need only 1000/30~33 IO
    - Necessary for efficient range / box queries
    - Desirable for nearest neighbor queries; not in this lecture
- Area of intensive research for decades

Caveats

- In commercial DBMS, multi-dimensional is supported for
  - Geometric objects: GIS extensions, spatial extender
  - Multimedia data (images, songs, …)
- Things get tricky if data is not uniformly distributed
  - Dependent / correlated attributes (age – weight, income, height)
  - Clustered values (e.g. population density)
  - Special distributions (normal, Zipf, …)
  - Skew – deviation from assumed distribution
- Curse of dimensionality: MDIS degrade with more dims
  - Trees difficult to balance, bad space usage, excessive management cost, expensive insertions/deletions, …

Content of this Lecture

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

Partitioned Hashing

- Let a_1 , a_2 ,…, a_d be the attributes to be indexed
- Define a hash function h_i for each a_i generating a bitstring
- Definition
  - Let $h_i(a_i)$ map each a_i into a bitstring of length b_i
  - Let $b=\sum b_i$ (length of global hash key in bits)
  - The global hash function $h(v_1, v_2, . . . , v_d) \rightarrow [0, ..., 2^b-1]$

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

Example

- Data: (3,6),(6,7),(1,1),(3,1),(5,6),(4,3),(5,0),(6,1),(0,4),(7,2)
- Let h_1, h_2 be ($b_1=b_2=1$, $b=2$)

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

- Four buckets with addresses 00, 01, 10, 11

a_1 \ a_2	0	1
0	(1,1) (3,1)	(3,6) (0,4)
1	(4,3)(5,0) (6,1) (7,2)	(6,7) (5,6)

- Note: This is an order preserving hash function – rare!
  - Modulo is not order preserving

Queries with Partitioned Hashing

- Exact queries: Direct access to bucket
  - All points in bucket are candidates; check identity to query
- Partial queries
  - Only parts of the global hash key are determined
  - Use those as filter; scan all buckets passing the filter
  - Let c be the number of unspecified bits
    - Then 2^c buckets must be searched
    - These are certainly not ordered on disk– random IO
- Range / box queries
  - Not efficiently supported if hash functions are not order preserving

Partitioned Hashing: Conclusions

- No adaptation to skew
  - Long overflow chains or large directories
- Size: Static size of hash table
  - Can only be saved by overflow chains
  - But: Can be combined with extensible/linear hashing
- Locality: Neighboring points in space not nearby in index
  - Usually, hash functions are not order preserving to achieve more uniform spread
  - Bad support for all non-exact queries or nearest neighbor queries

Content of this Lecture

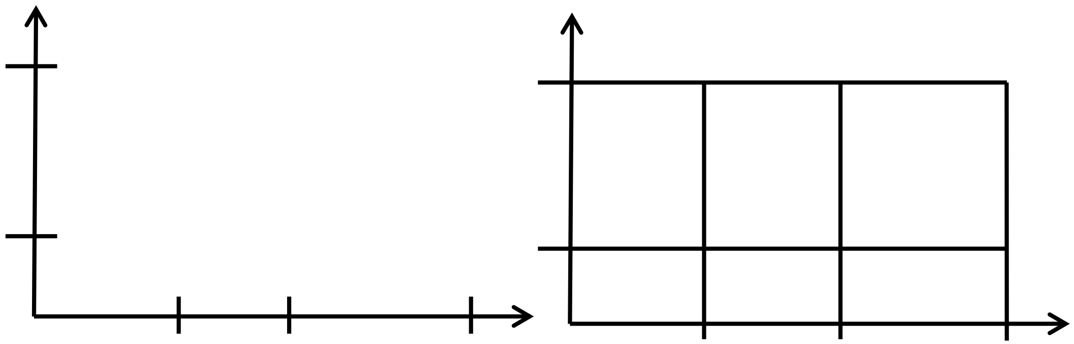
- Introduction to multidimensional indexing
- Partitioned Hashing
- Grid Files
- kdb Trees
- R Trees

Grid File

- Classical multidimensional index structure
  - Nievergelt, J., Hinterberger, H. and Sevcik, K. C. (1984). "The Grid File: An Adaptable, Symmetric Multikey File Structure." *ACM TODS*
  - Can be seen as extensible version of partitioned hashing
  - Good for uniformly distributed data, bad for skewed data
  - Numerous variations, we only look at the basic method
- Design goals
  - Aims to support all types of queries
  - Guarantee “two IO” access to each point
    - Under certain assumptions
  - Adapt dynamically to the number of points

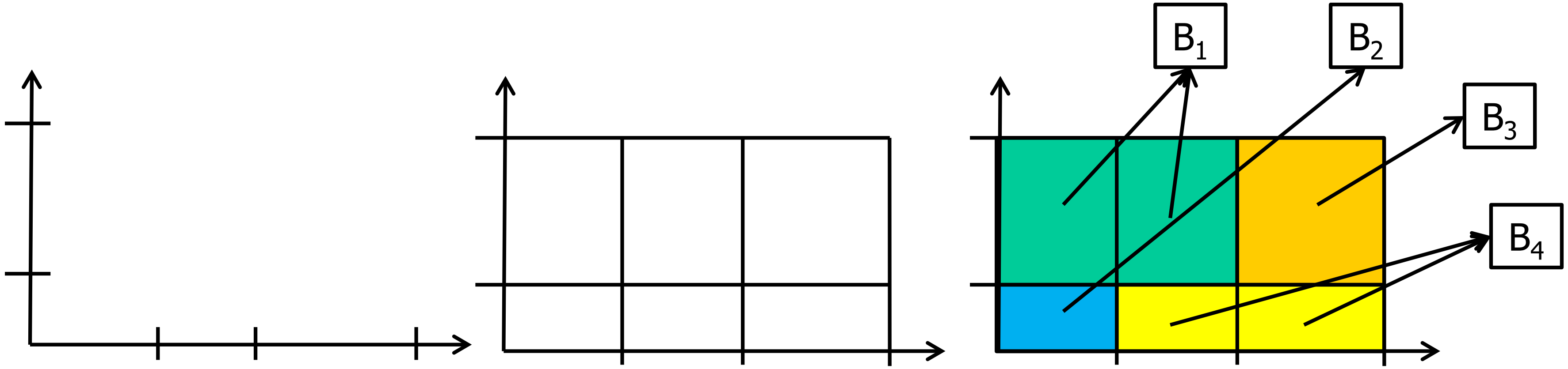
Principle

- Partition each dimension into disjoint intervals (scales)
  - EXCESS: Uniform scales; less adaptive, no scale management
- Intersection of all intervals defines grid cells
  - d-dimensional hypercubes
- Grid cells are addressed from the grid directory (GD)
  - A simple multidimensional array



Principle

- Partition each dimension into disjoint intervals (scales)
- Intersection of all intervals defines grid cells
- Grid cells are addressed from the grid directory (GD)
- Cells are grouped in regions; region = bucket = block
  - When multi-cell region overflows – split
  - When single-cell region overflows – new scale, change GD
- Buckets hold values + TID



Exact Queries

- Assumption: GD in main memory
  - Size: $|S_1|*|S_2|*\ldots|S_d|$, when S_i is the set of scales for dimension I
  - Becomes (too) large for high dimensional data

- 1. Compute grid cell
  - Look-up coordinates in scales to obtain GD coordinates
    - E.g. binsearch on sorted scale list
  - Cell in GD contains pointer to region/bucket on disk
  - Bucket contains all data points in this grid cell (maybe more)
- 2. Load bucket and find point(s): 1st IO
  - As usual, we do not look at how to search inside a bucket
- 3. Access record following TID: 2nd IO

Other Queries

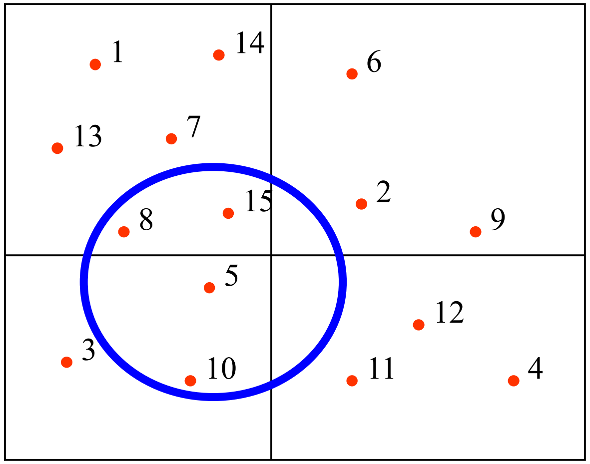
- Box queries
  - Compute all matching scales
  - Access all corresponding cells in GD
  - Load and search all buckets
- Partial queries
  - Compute partial GD coordinates
  - All GD cells with these coordinates may contain points
- Both cases: Efficiency depends on matching of range conditions to scales
  - Using scales as range conditions – very efficient
  - Using range conditions in between scales – less efficient

Excursion: Nearest Neighbor Queries

- Find bucket containing query point
- Search points in this region and choose closest
  - Can we finish with the closest point in this region?

Nearest Neighbor Queries

- Find bucket containing query point
- Search points in this region and choose closest
  - Can we finish with the closest point in this region?
  - Usually not
    - Check distances to all borders
    - If point found is closer than any border, we are done
    - Otherwise, we need to search neighboring regions
    - Do iteratively and always adapt radius to current closest point
    - Visit neighbor buckets in order of distance to query point
  - Very fast if nearest neighbor provably is in same region

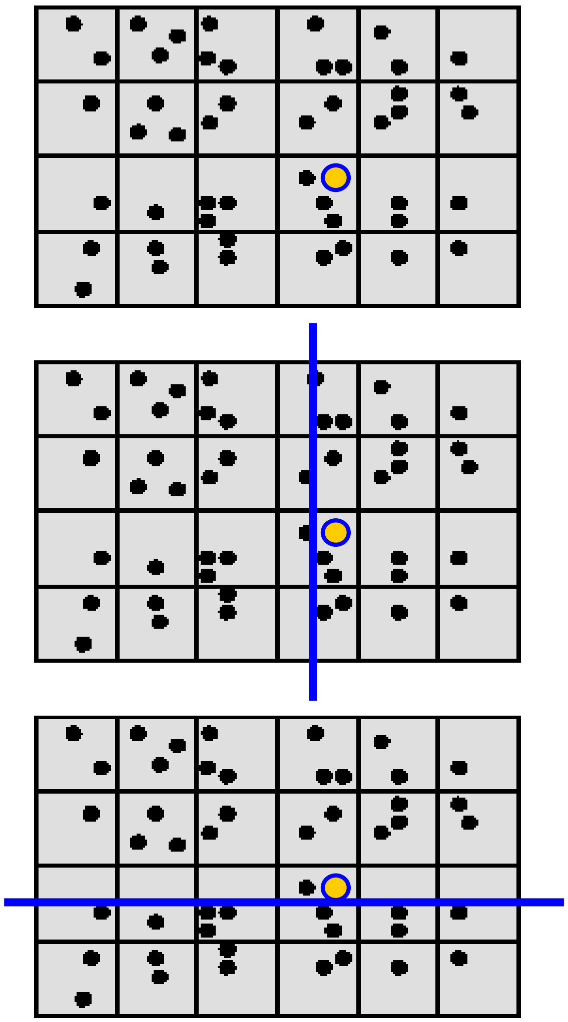


Inserting Points

- Search grid cell; if bucket has space: Insert point
- Otherwise (overflow): Split
  - Assume we have to split a single-cell region
  - Choose a dimension and new scale within region interval
  - Split all affected GD cells – cuts through all dimensions
    - Consider n dimensions and S_i scales in dimension i
    - Split in dim i affects $d_1 * \ldots * d_{i-1} * d_{i+1} * \ldots * d_n$ cells in GD
    - Example: d=3, $S_i=4$; $|GD|=4^3=64$; any split affects 4^2 cells
  - Split overflown bucket along new scale (new region)
  - Do not split other (un-overflown) buckets containing the new scale
    - Only copy pointers within GD
  - Choice of dimension and interval is difficult
    - Optimally, we would like to "split" many rather full blocks
    - We also want to consider our future expectation

Example

- Imagine one block holds 3 points
  - [Usually scales are unevenly spaced]
- 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
- Note: Real splits will happen only in the future



Choosing a Split

- We wish
  - W1: Split points evenly in overflow bucket
  - W2: Future-Split points evenly other affected buckets
  - W3: Split future points within bucket range evenly
  - W4: Future-Split future points within other affected buckets
- W1: Sort points in every dimension and chose median
- W2 is expensive: Load all affected blocks in every dim.
- W3, W4: Require guessing the future
  - W1 and W2 assume that future distribution is same as past dist.
- Wishes can be are contradicting
  - A balanced split in overflown cells (W1) may lead to unbalanced splits in other cells (W2)
- Alternative: Round-robin in dimensions and chose median

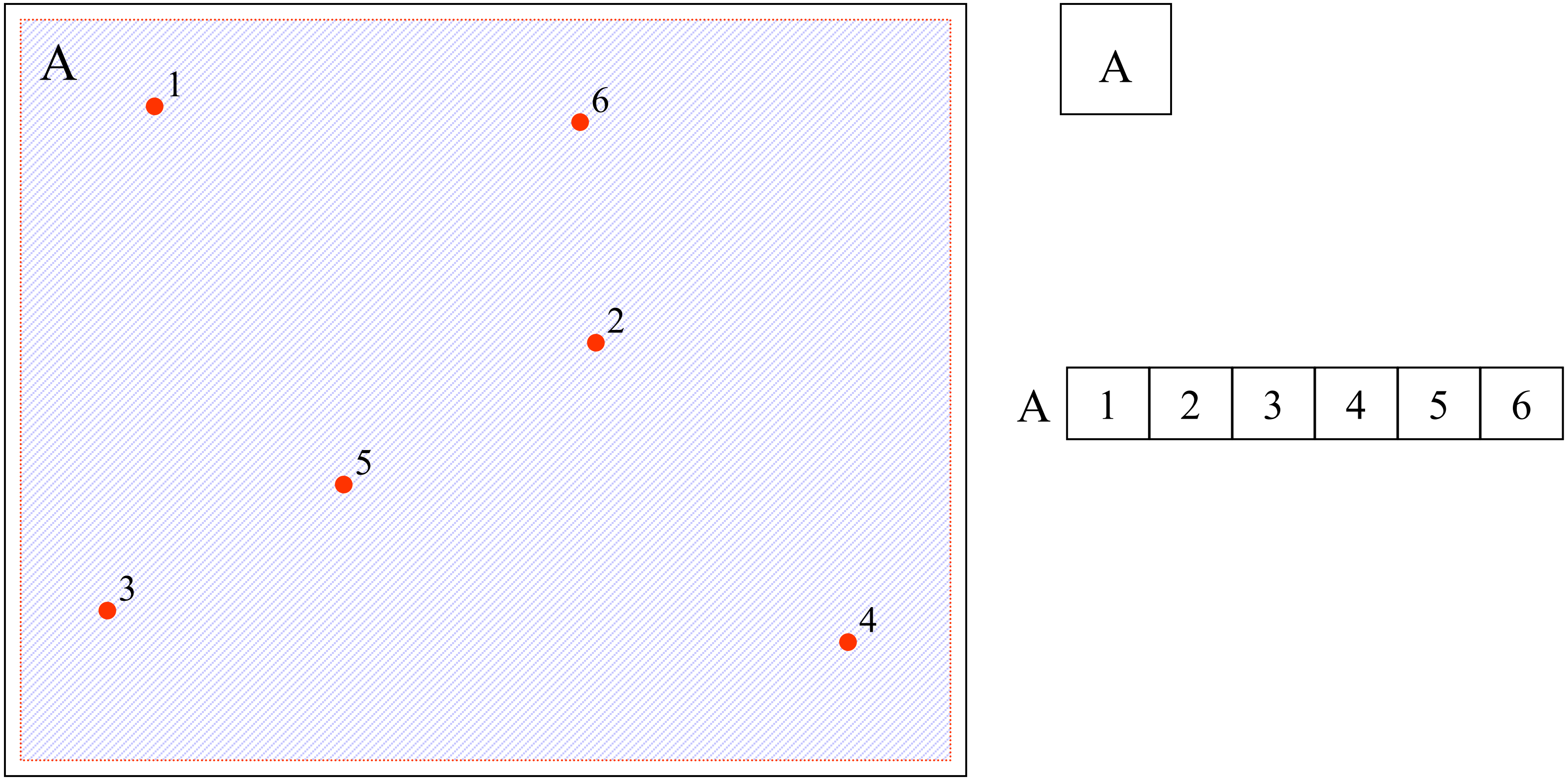
Inserting Points in Multi-Cell Regions

- Overflow in a multi-cell region
  - A bucket to which multiple GD entries point
- Split region into smaller regions (or cells) along existing, not yet realized scales
  - GRID file only considers existing scales not yet used for split in this region
    - No local adaptation – decisions from the past have to be obeyed
  - GD structure is left unchanged; only cell entries change
- Which scale to use (there may be more than one)?
  - This is a local decision
  - Chose splits that best distribute the bucket that is split

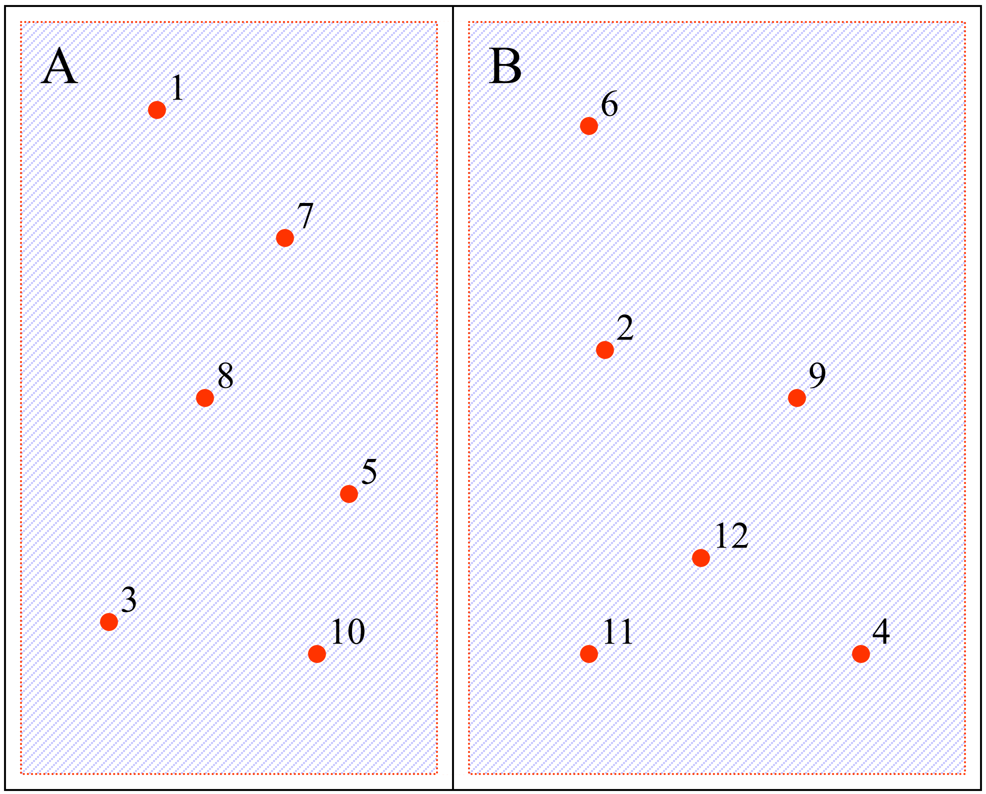
???

Grid File Example 1 [J. Gehrke]

Assume k=6



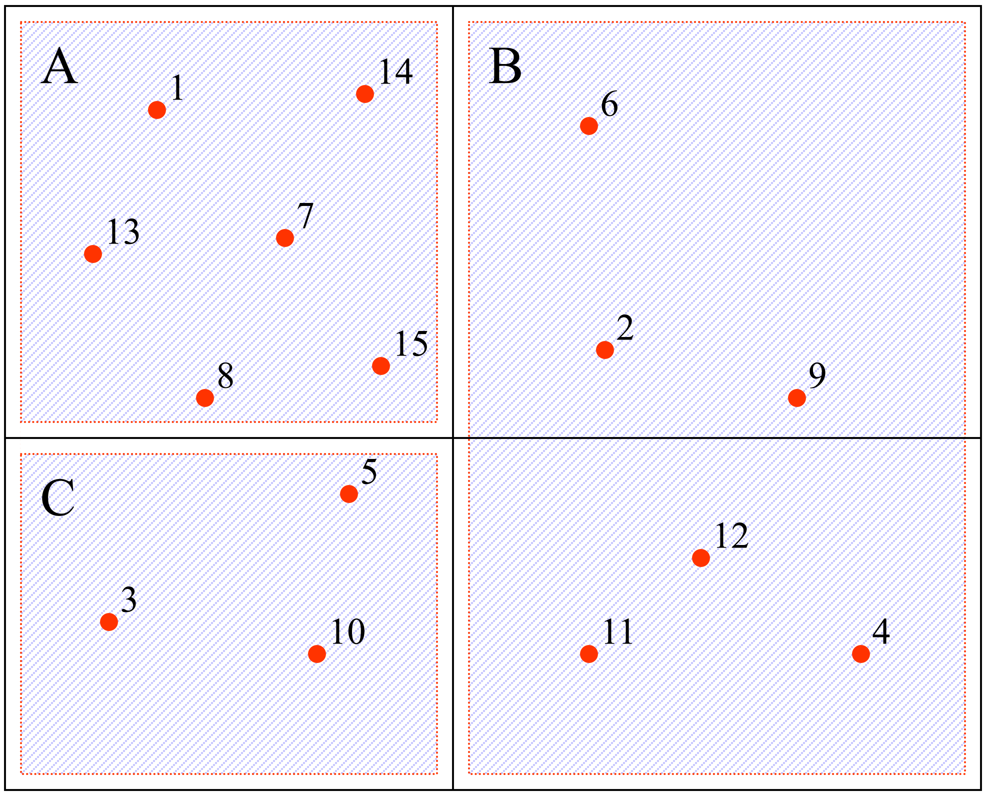
Grid File Example 2



A	B

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

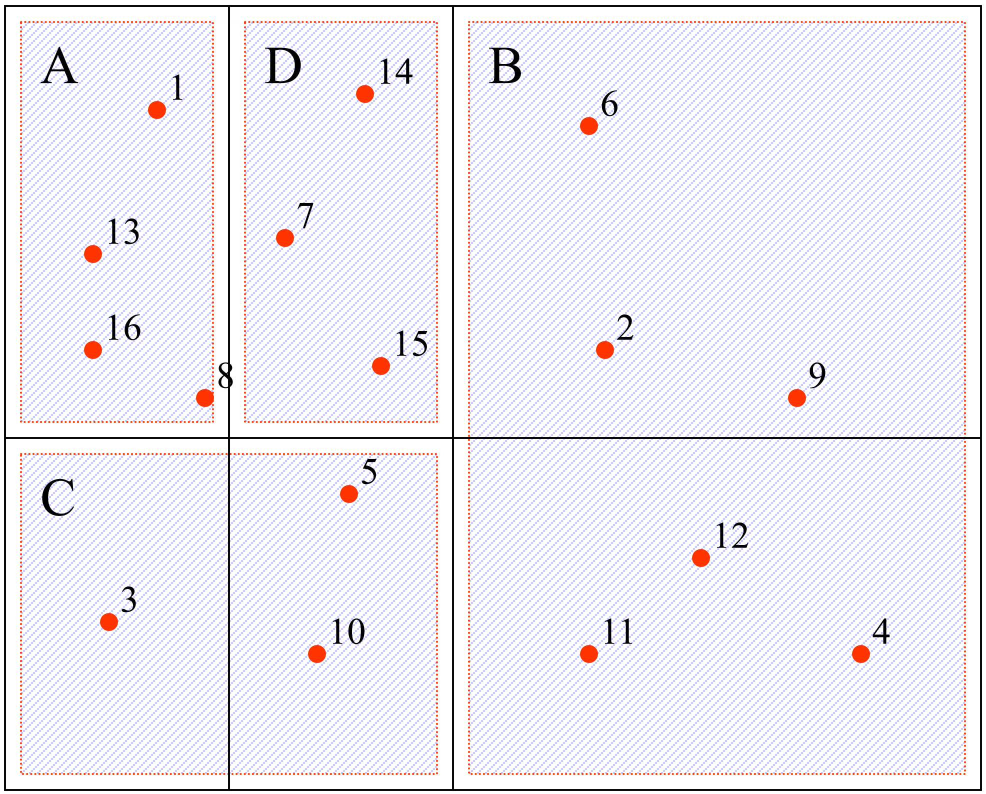
Grid File Example 3



A	B
C	B

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

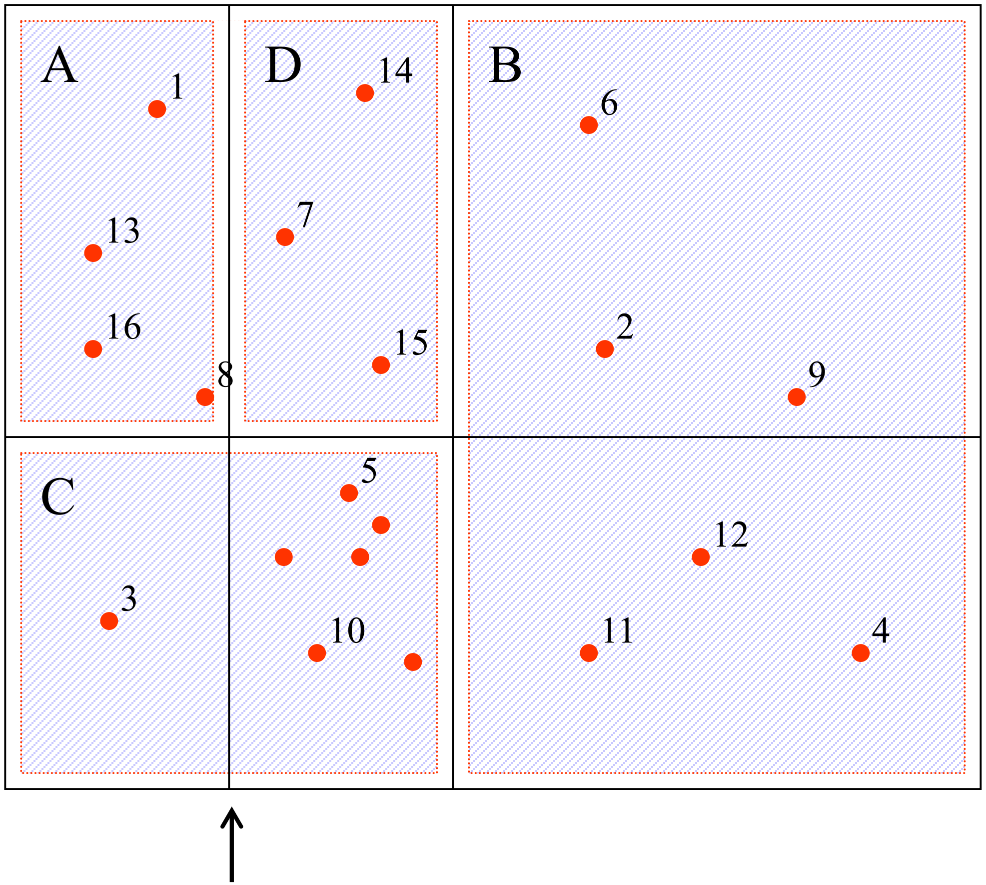
Grid File Example 4



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			

One Future

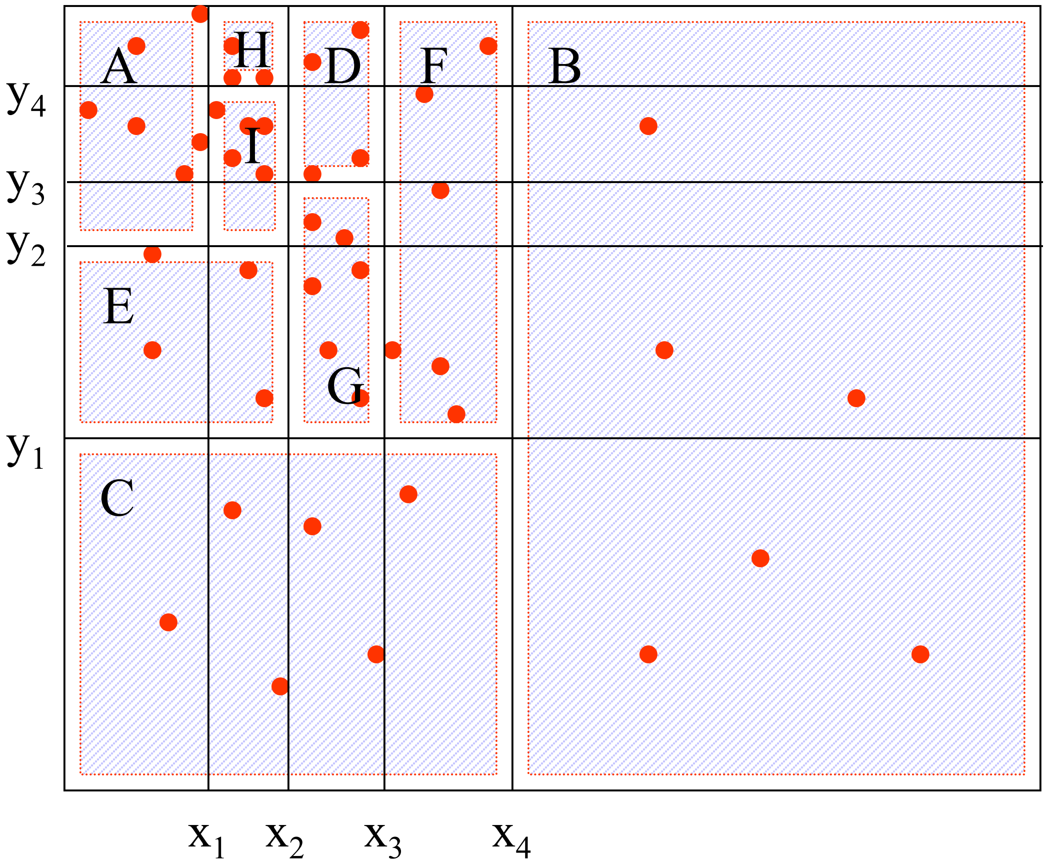


A	D	B
C	C	B

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

We now must perform this split; creates one almost empty and one full bucket; next split will happen soon

Grid File Example 5



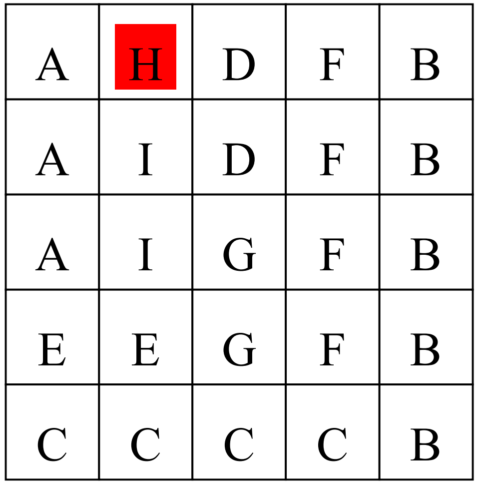
A	H	D	F	B
A	I	D	F	B
A	I	G	F	B
E	E	G	F	B
C	C	C	C	B

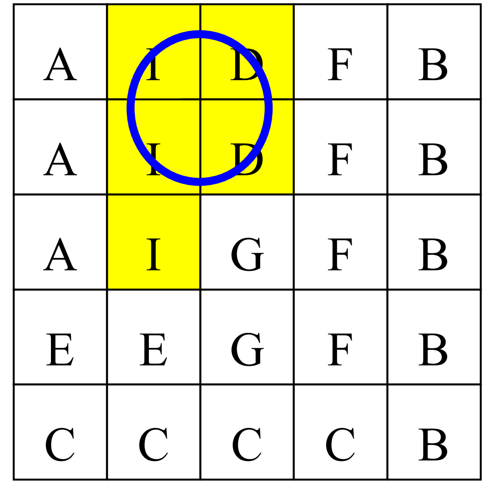
Deleting Points

- Search point and delete
- If bucket becomes ”almost empty”, try to merge with other buckets
  - A merge is the removal of a split – chose scale to “unmake”
  - Should build larger convex regions
  - This can become difficult
    - Potentially, more than two regions need to be merged to keep convexity
  - Eventually, also scales may be removed
    - Shrinkage of GD
  - Example: Where can we merge?

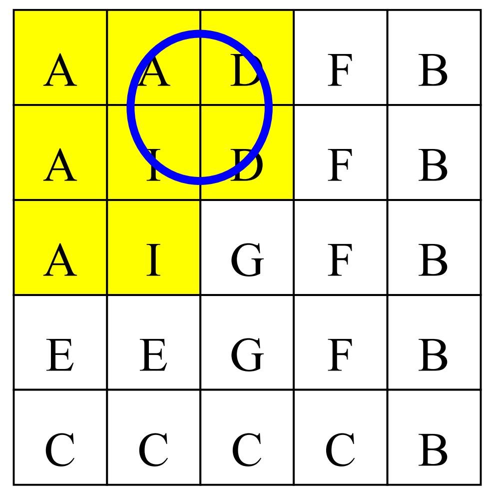
A	H	D	F	B
A	I	D	F	B
A	I	G	F	B
E	E	G	F	B
C	C	C	C	B

Convex Regions





- Non-convex regions: Range and neighborhood queries have to scan increasingly many buckets



Some Observations

- Grid files always split at hyperplanes parallel to the dimension axes
  - This is not always optimal
  - Use other bounding shapes: circles, polygons, etc.
  - More complex– forms might not disjointly fill the space any more
  - Allow overlaps (see R trees)
- There is no guaranteed block-fill degree – degeneration
- Choosing a new scale is a local decision with global consequences
  - No local adaptation: GD grows very fast
  - Need not be realized immediately, but restricts later choices in other regions
  - Bad adaptation to skewed data