Database Systems

WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Topics (2/6)

- indexing
  - one-dimensional
  - tree-structured
  - partition-based indexing
  - bulk-loading
  - main-memory indexing
  - hash-indexes
  - multi-dimensional indexes
  - differential indexing
  - read-optimized indexing
  - write-optimized indexing
  - data warehouse indexing
  - text indexing: inverted files
  - (flash-indexing)

Multi-dimensional Indexes.

Multi-dimensional Indexes.

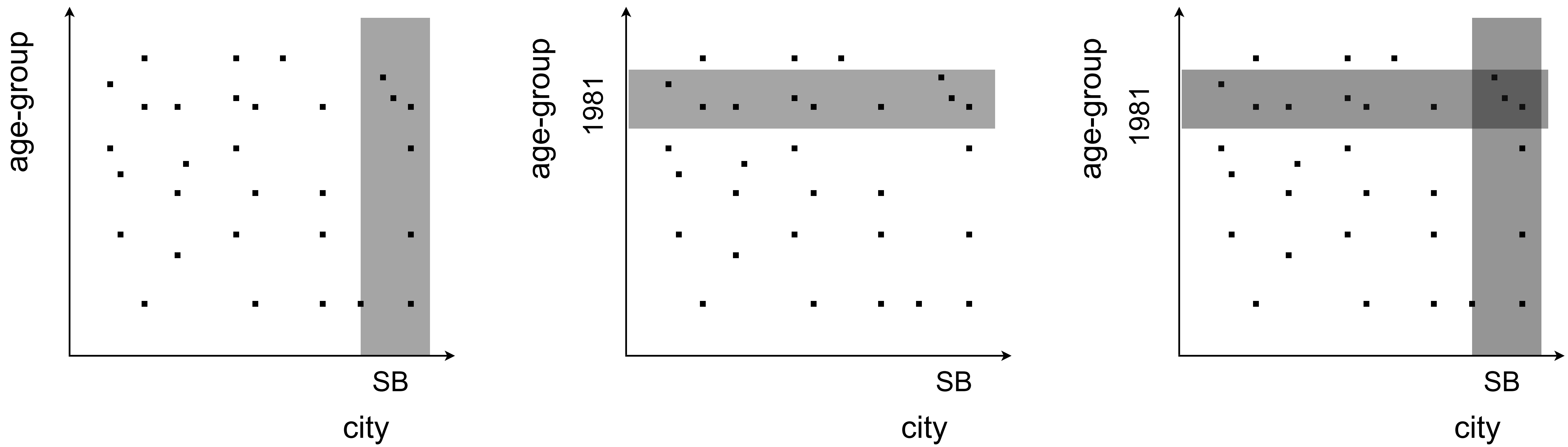
Introduction.

Motivation

- Queries treated in the previous section:
  - What is the address of the student having ID 424342?
  - Which students attend less than 2 lectures in this semester?
  - Which students live in Saarbrücken?
- How do we process the following queries:
  - Which students attend less than two lectures **AND** live in Saarbrücken?
  - Which students live in Saarbrücken **AND** were born in 1981?
- Problem: more than one attribute per query
- How do we index this?

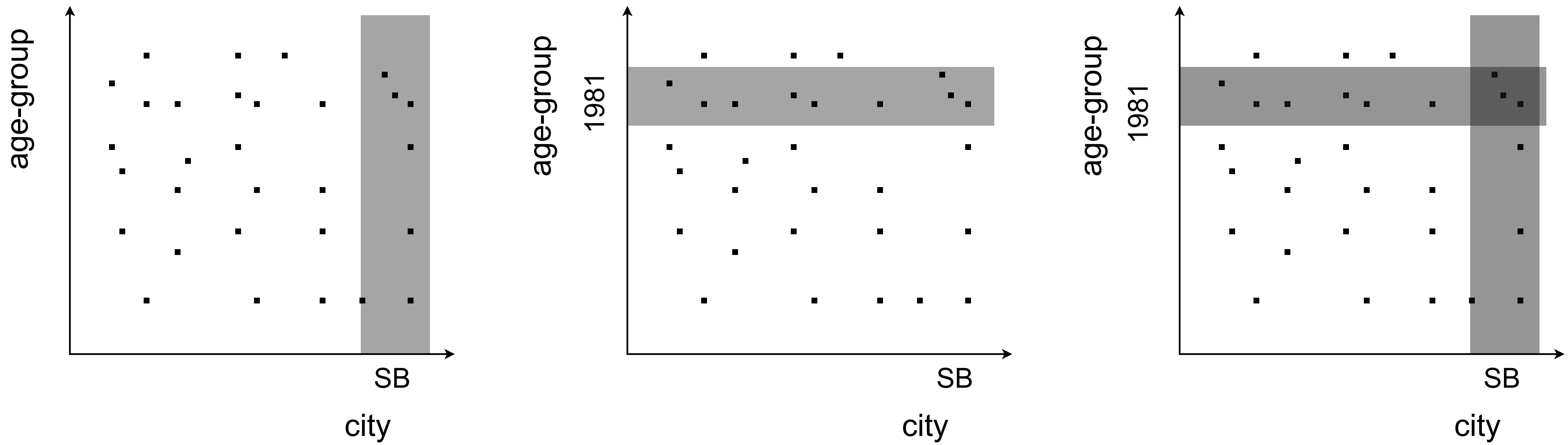
Motivation

- Which students live in Saarbrücken AND were born in 1981?



- 1. Approach:
  - one secondary index on 'city'
  - one secondary index on 'age-group'
  - compute intersection on TID-lists

Intersection of TID-Lists



- Discussion:
  - for each index access log N random accesses + cost for result TID-lists,
    in total: #attributes * (log N accesses + cost for result TID-lists)
  - high cost for intersection if one of the indexes delivers many results
- But:
  - this approach may work well in many situations

Key Concatenation

- 2. Approach:
  - treat ‘city’ + ‘age-group’ as a concatenated key
  - create a secondary index using the concatenated key
- example:
  - index.find_key(‘SB1981’)
  - index.find_key(‘SB1984’)
- Discussion:
  - cost equivalent to the cost of a one-dimensional access!

Key Concatenation

- What happens to one-dimensional queries?
  - index.find_key('SB????')
    - no problem
    - first: find_key using prefix 'SB',
    - then: ISAM
  - index. find_key('??1984')
    - index useless
    - either: generate all possible prefixes and search (very inefficient)
    - or: scan all entries (FTS, full-table scan)

Key Concatenation

- Only efficient if a prefix of the key is specified in the query
- Data is clustered **lexicographically** in attribute order in the index

A (1. sort-attribute)	B (2. sort-attribute)	C (3. sort-attribute)
A1	B1	C1
A1	B1	C2
A1	B2	C1
A1	B2	C2
A2	B1	C1
A2	B1	C2

- Clustering of data useless for queries like
  - B=42
  - C=55
  - B=42 $\wedge$ C=55
  - A=42 $\wedge$ C=53

Requirements for ‘Real‘ Multi-dimensional Indexes

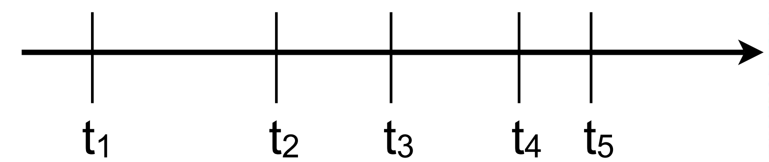
- Main goal/requirement:
  - physical clustering of nearby data, i.e.,
    data that is close in the n-dimensional space should also be close on the index
- Other requirements:
  - similar to one-dimensional index structures

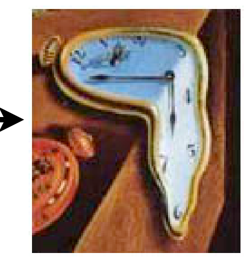
Different Types of Data

- relational data

(42, Hugo, Müller)

- temporal data
  - historical databases
  - data warehouses





- geographical data
  - GIS (e.g., Google Maps)
  - location based services

- high-dimensional data
  - feature-vectors
  - data mining
  - Example: similarity-search on images

$\vec{x}=(0.1,0.8,0.3,0,2,0,5,0.8)$

Query Types

d: #dimensions

- **Exact Point Query**
  specifies an equals predicate for all d dimensions:
  $Q=(A_1=a_1) \wedge (A_2=a_2) \wedge \ldots \wedge (A_d=a_d)$
- **Partial Point Query**
  specifies an equals predicate for s<d dimensions:
  $Q=(A_{i1}=a_{i1}) \wedge (A_{i2}=a_{i2}) \wedge \ldots \wedge (A_{is}=a_{is})$
  where $1 \leq s \leq d$ and $1 \leq i_1 < i_2 < \ldots < i_s \leq d$
- **Exact Range Query**
  specifies an interval predicate r_i=[min;max] for all dimensions:
  $Q=(A_1 \cap r_1 \neq \emptyset) \wedge (A_2 \cap r_2 \neq \emptyset) \wedge \ldots \wedge (A_d \cap r_d \neq \emptyset)$
- **Partial Range Query**
  specifies an interval predicate r_i=[min;max] for s<d dimensions:
  $Q= (A_{i1} \cap r_{i1} \neq \emptyset) \wedge (A_{i2} \cap r_{i2} \neq \emptyset) \wedge \ldots \wedge (A_{is} \cap r_{is} \neq \emptyset)$
  where $1 \leq s \leq d$ and $1 \leq i_1 < i_2 < \ldots < i_s \leq d$

General Query

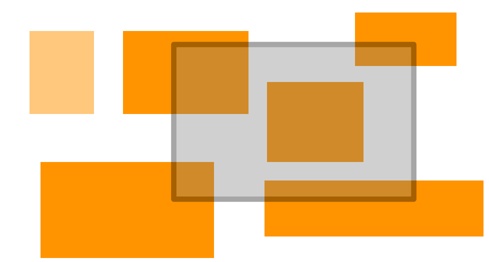
d: #dimensions

General Query:
specifies an interval predicate ri=[min;max], min≤max for $s \leq d$ dimensions
$$Q = (A_{i1} \cap r_{i1} \neq \emptyset) \wedge (A_{i2} \cap r_{i2} \neq \emptyset) \wedge \ldots \wedge (A_{is} \cap r_{is} \neq \emptyset)$$
where $1 \leq s \leq d$ and $1 \leq i_1 < i_2 < \ldots < i_s \leq d$

- Discussion:
  - min=max possible
  - easy to extend for half-open or open intervals

Intersection vs. Containment

- Intersection:
  - for each dimension a non-empty intersection of result and query interval is enough to fullfill the predicate
  - $A_{i1} \cap r_{i1} \neq \emptyset$



- Containment:
  - for each dimension the result interval has to be contained in the query interval
  - $A_{i1} \supseteq r_{i1}$

Nearest-Neighbor Query

- **find_NN(x)** given a data set Y and a record x retrieves the record y from Y having the smallest distance to x
- distance defined by a metric
- Examples:

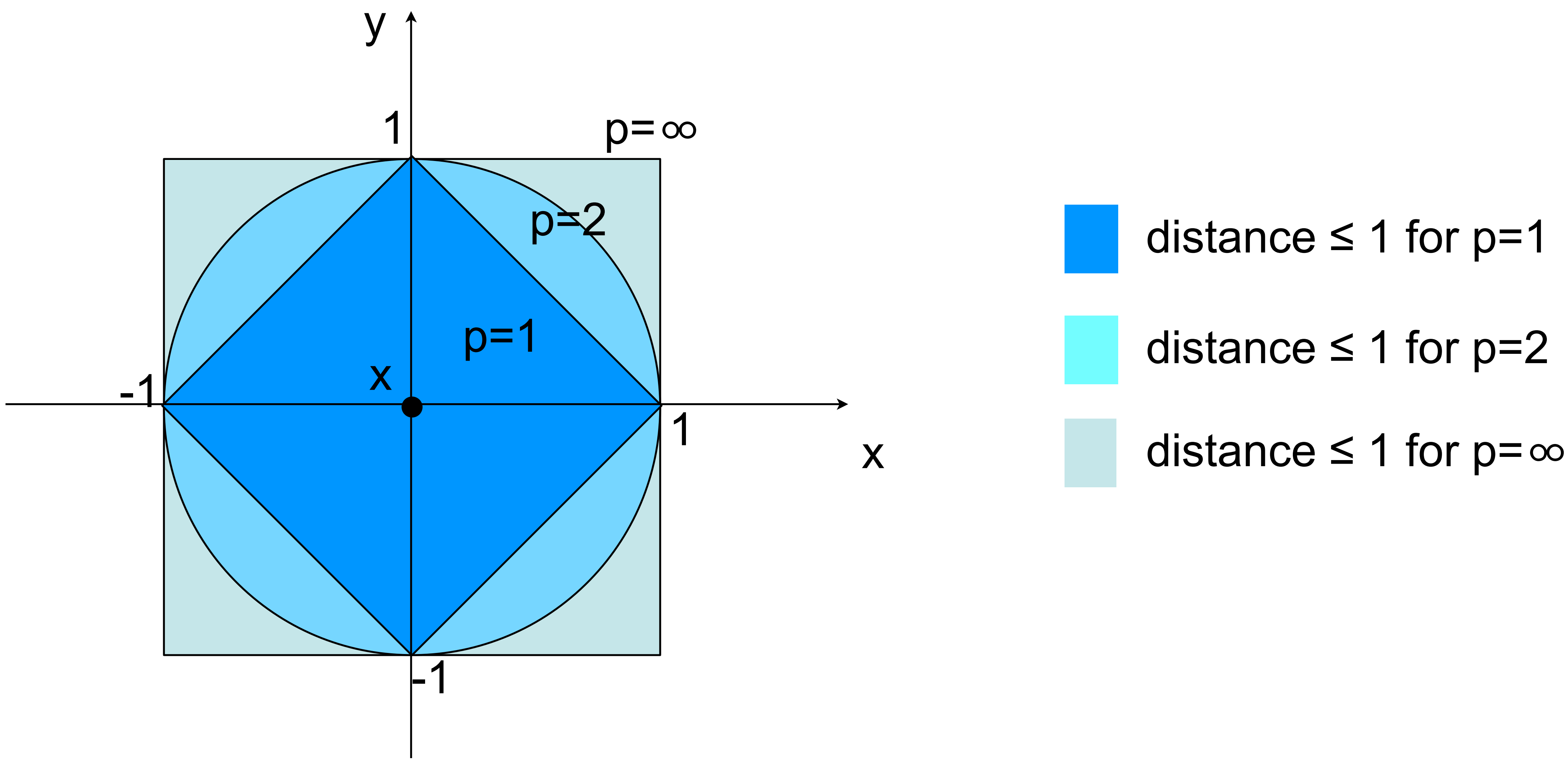
$$\text{distance} = \sqrt[2]{\sum_{i=1}^{d} |x_i - y_i|^2} \qquad \text{(euclidean metric)}$$

$$\text{distance} = \sum_{i=1}^{d} |x_i - y_i| \qquad \text{(manhatten distance metric)}$$

$$\text{distance} = \sqrt[p]{\sum_{i=1}^{d} |x_i - y_i|^p} \qquad \text{(p-metric)}$$

- Example query: Where is the next pizzeria?

p-metric



$$\text{distance} = \sqrt[p]{\sum_{i=1}^{d} |x_i - y_i|^p} \qquad \text{(p-metric)}$$

It holds: $\sqrt[p]{\sum_{i=1}^{d} |x_i - y_i|^p} \leq \sqrt[q]{\sum_{i=1}^{d} |x_i - y_i|^q}$ for $p \geq q$

k-Nearest-Neighbor Query

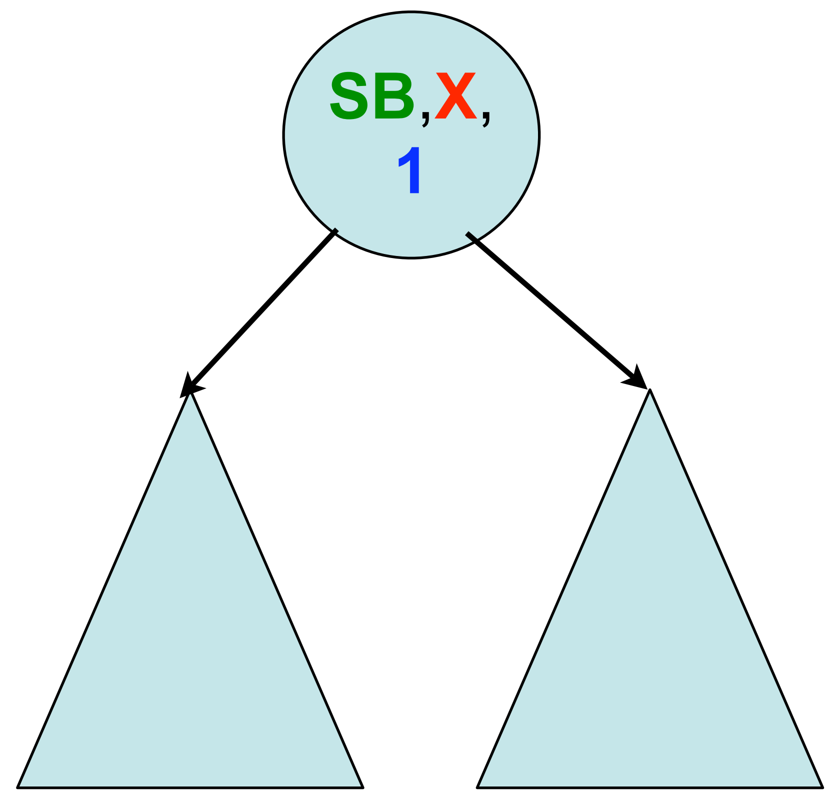
- generalization of nearest-neighbor query
- 1-NN = NN
- find_kNN(x)
  given a data set Y returns k records Y'=<y1,...,yk> from Y having the k-smallest distance to x,
  - result delivered in order y1,...,yk
- Example query:
  Which are the three closest pizzerias?

Multi-dimensional Indexes.

Digital Trees.

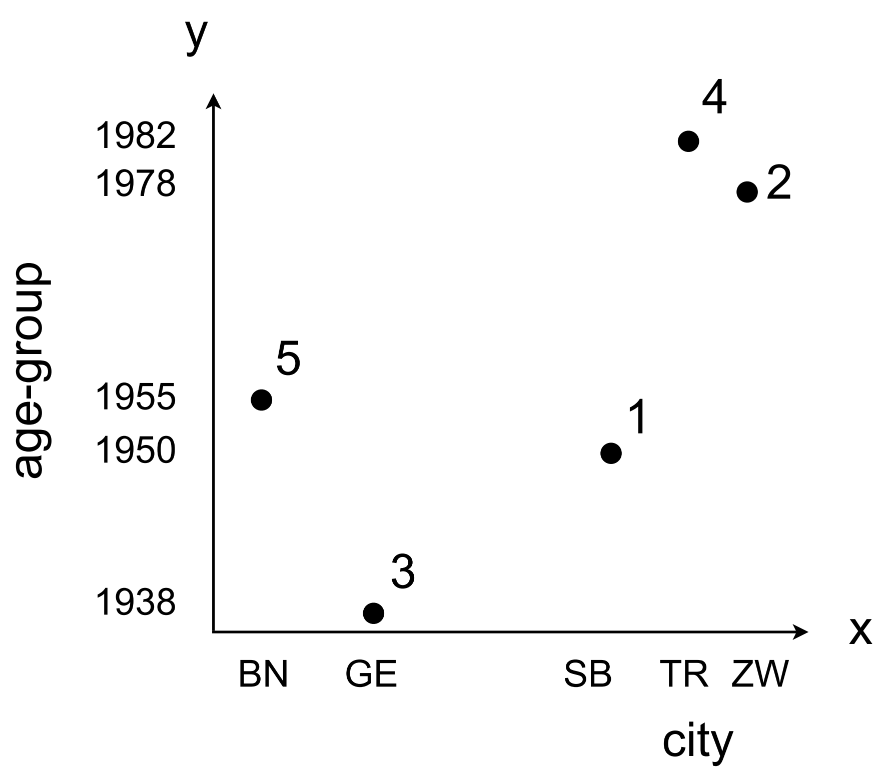
A Simple Multi-Dimensional Index

- name: kd-**tree**
- core idea: extend binary tree
- many variants exist
- node layout:
  - left child
  - pivot, pivot-dimension, value
  - right child



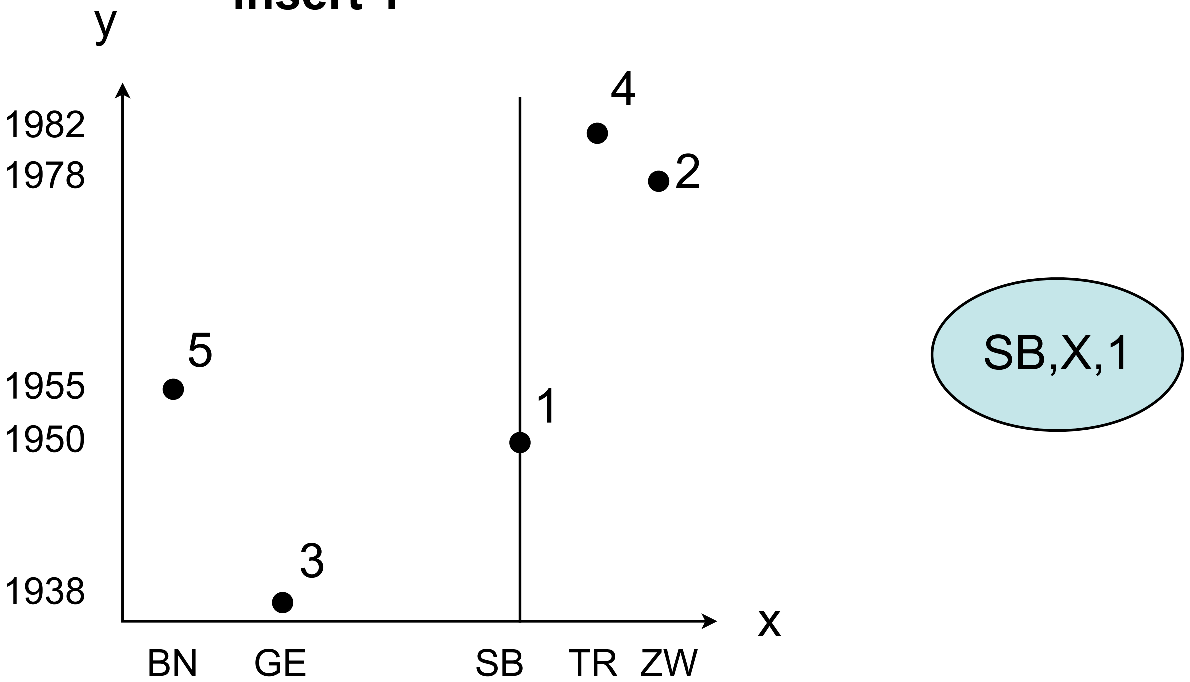
Example:

1: (Wozniak, 1950, SB)
2: (Müller, 1978, ZW)
3: (Knuth, 1938, GE)
4: (Meier, 1982, TR)
5: (Jobs, 1955, BN)

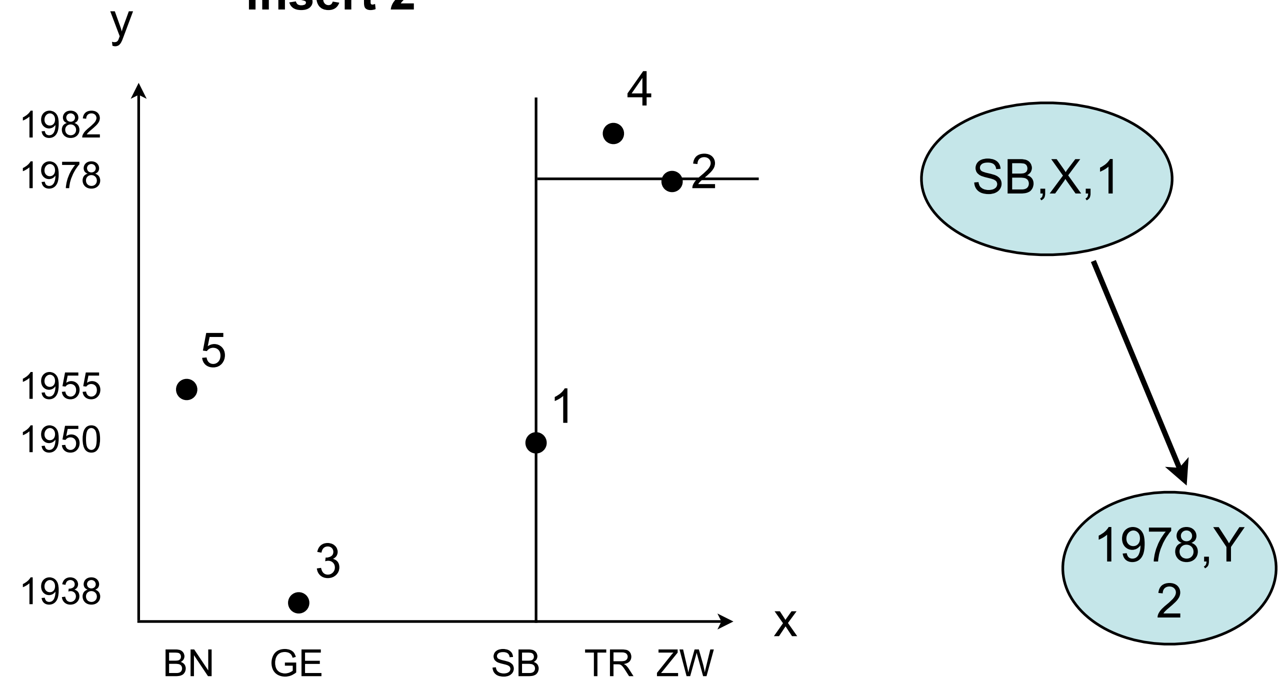


Tree Example

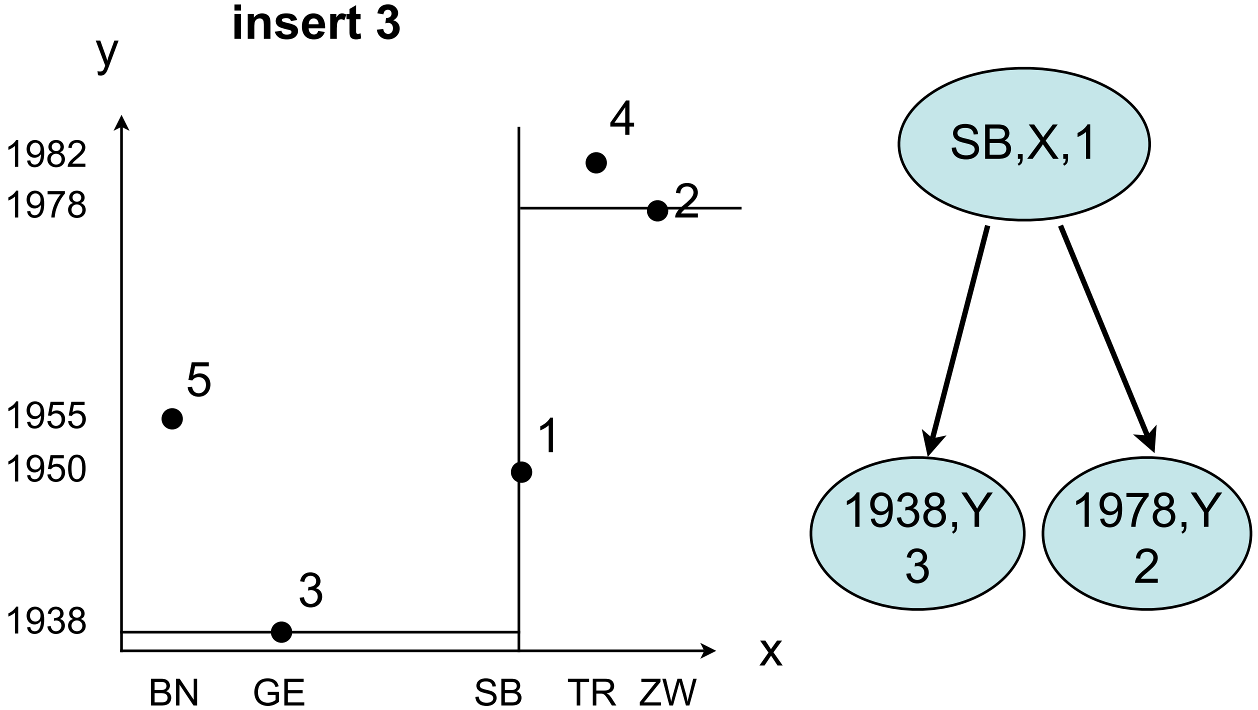
insert 1



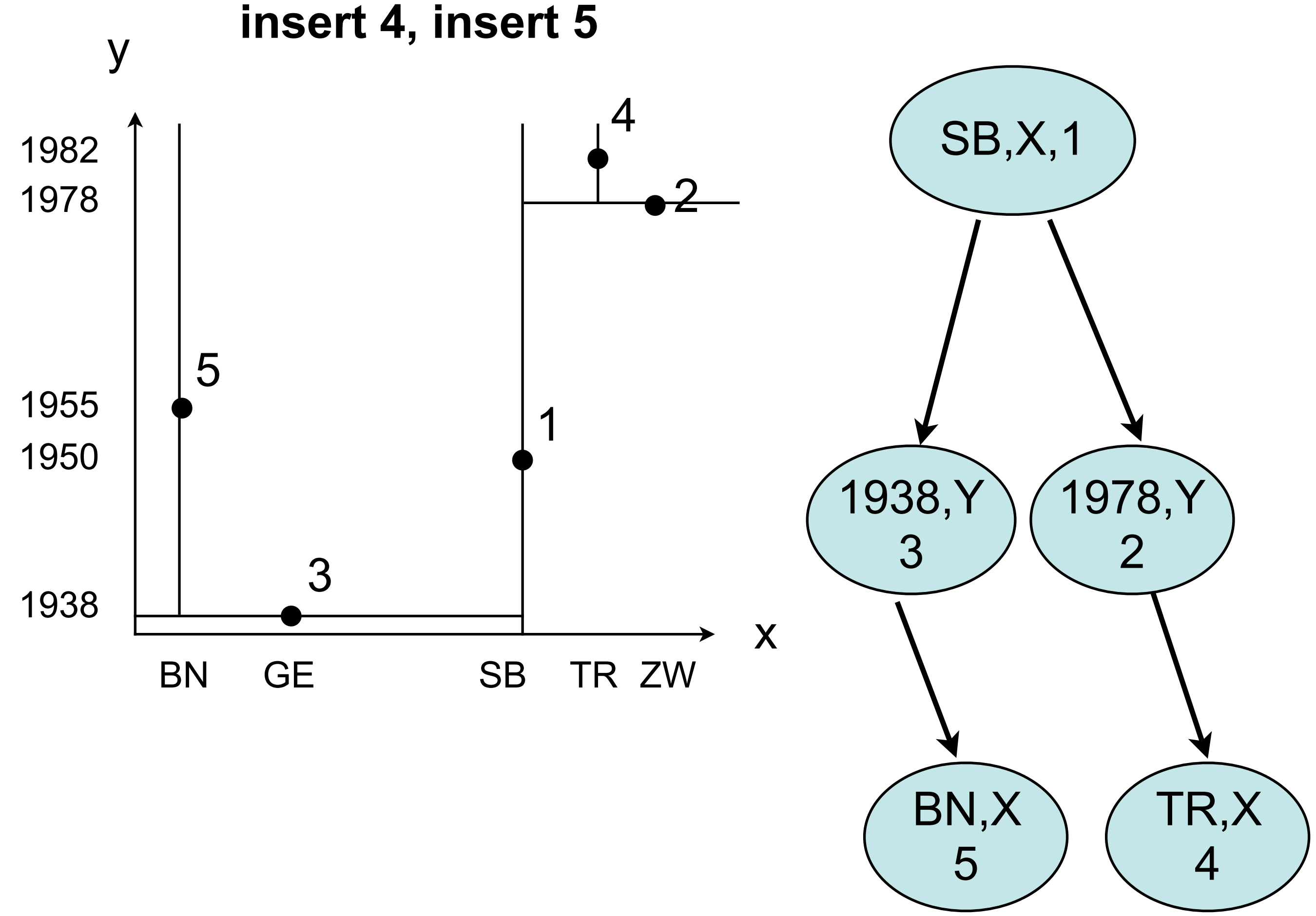
insert 2



insert 3

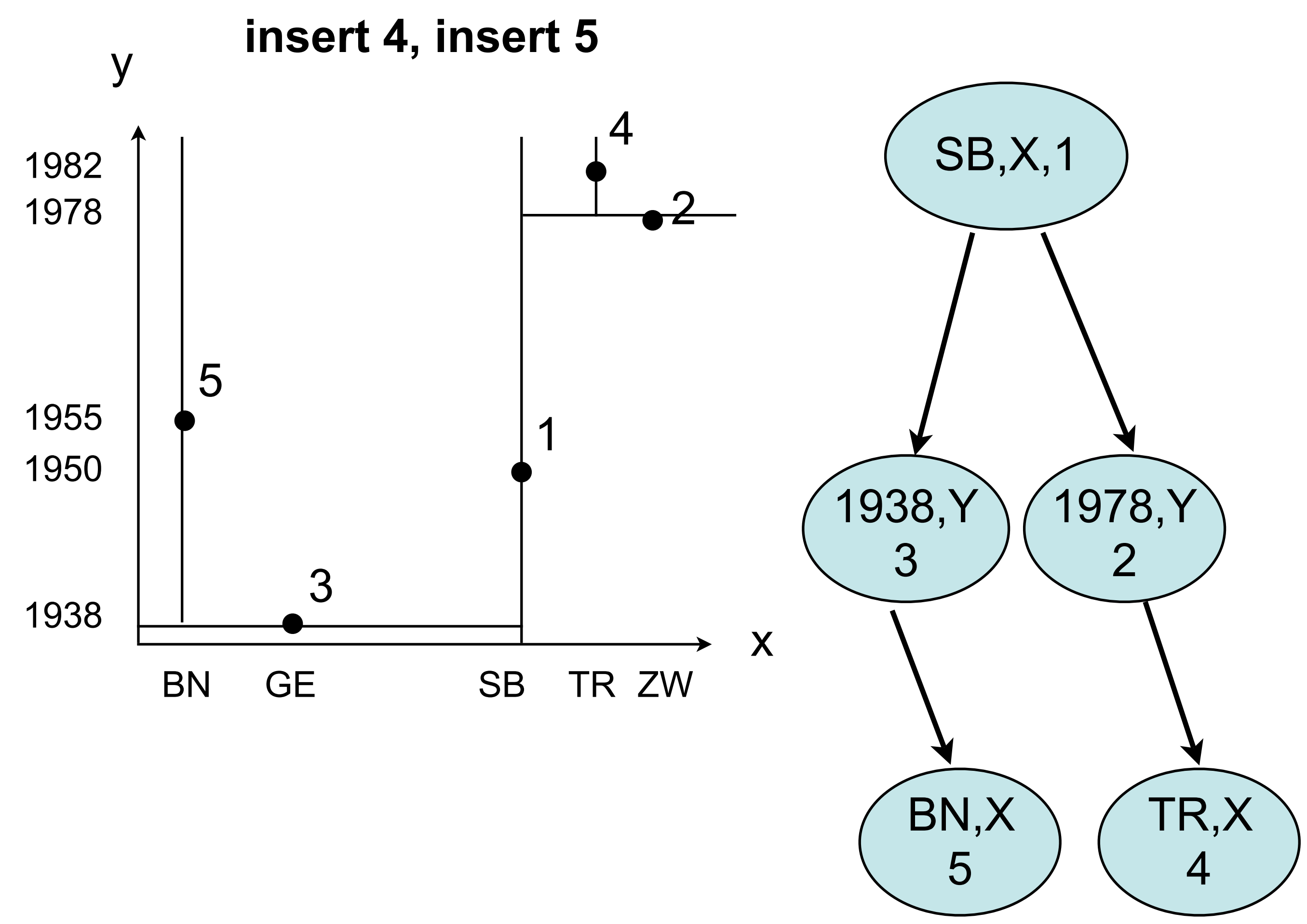


insert 4, insert 5



Discussion

- order of insertions determines partitioning function of the data space
- tree not balanced
- skewed data distributions may degenerate tree into a list
- the partitioning of the data space is determined by **the data**
- also: **order** of insertions determines partitioning



kd-Trie

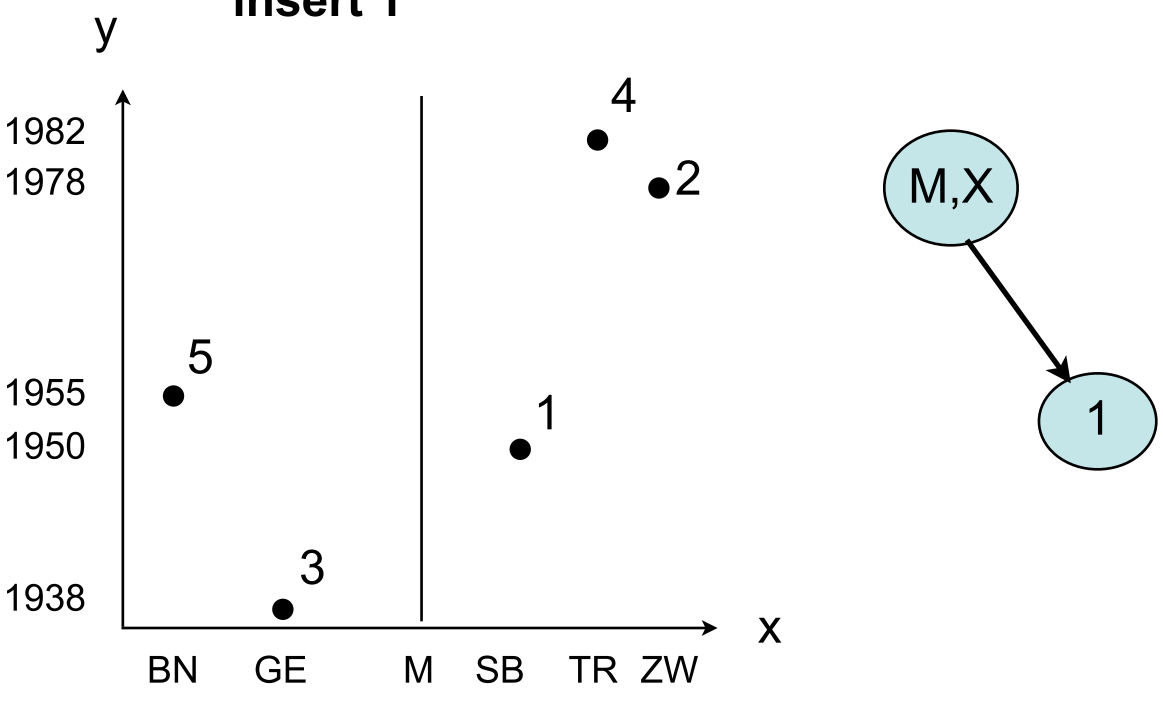
- like kd-tree
- But: the partitioning of the data space is **independent from the data**

Tree	Trie
partition based on data	partition independent from data

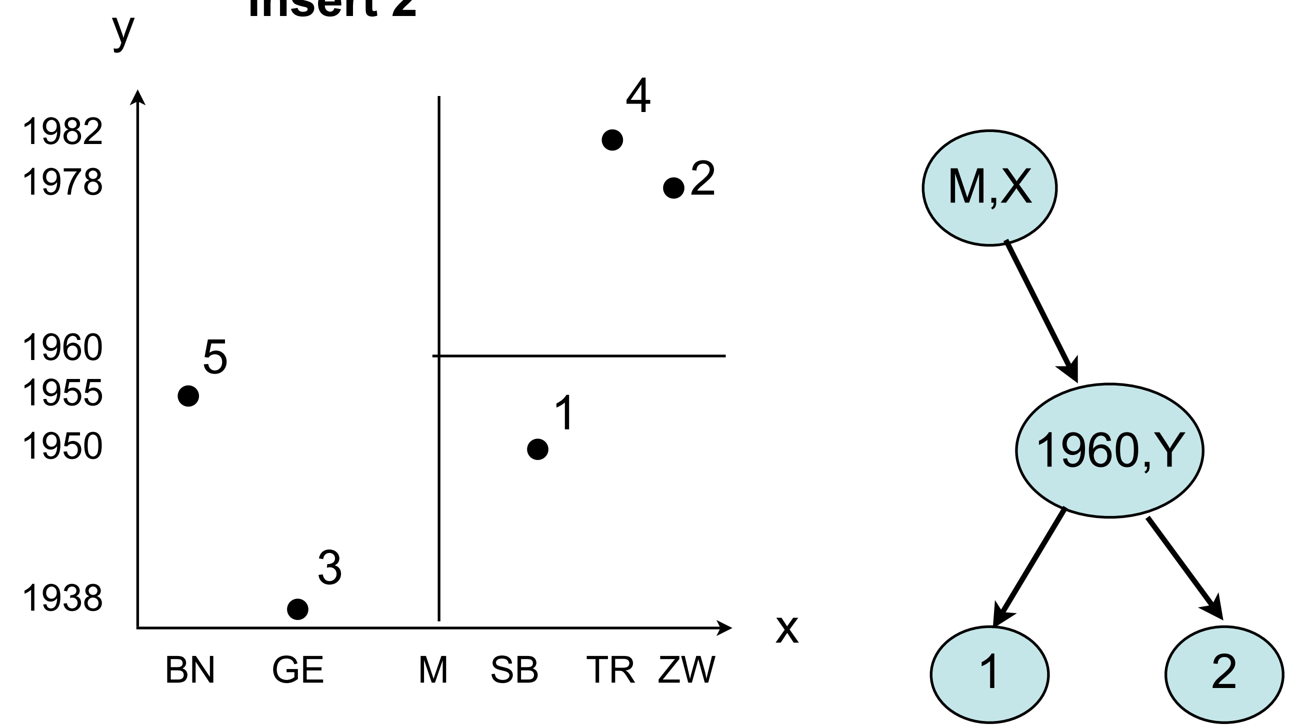
- **Note:**
  This is not uniformly used throughout literature:
  Many ‘tries’ are called ‘tree’!

Trie Example

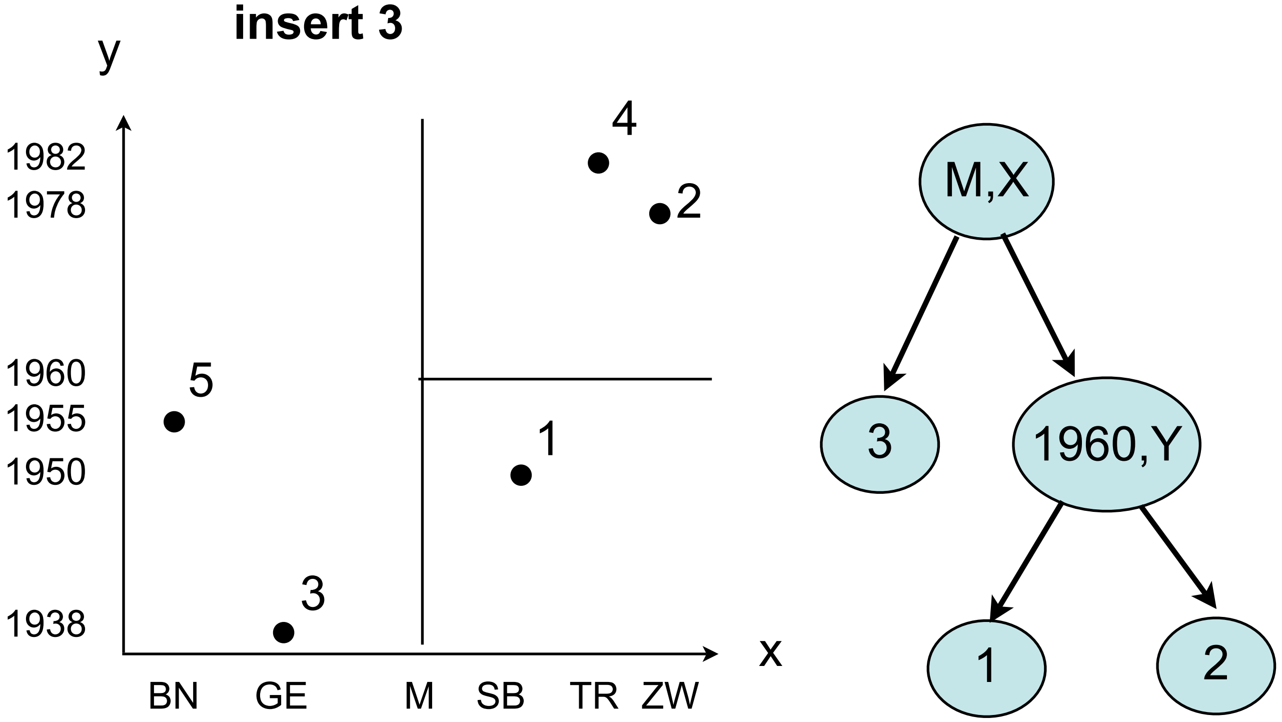
insert 1



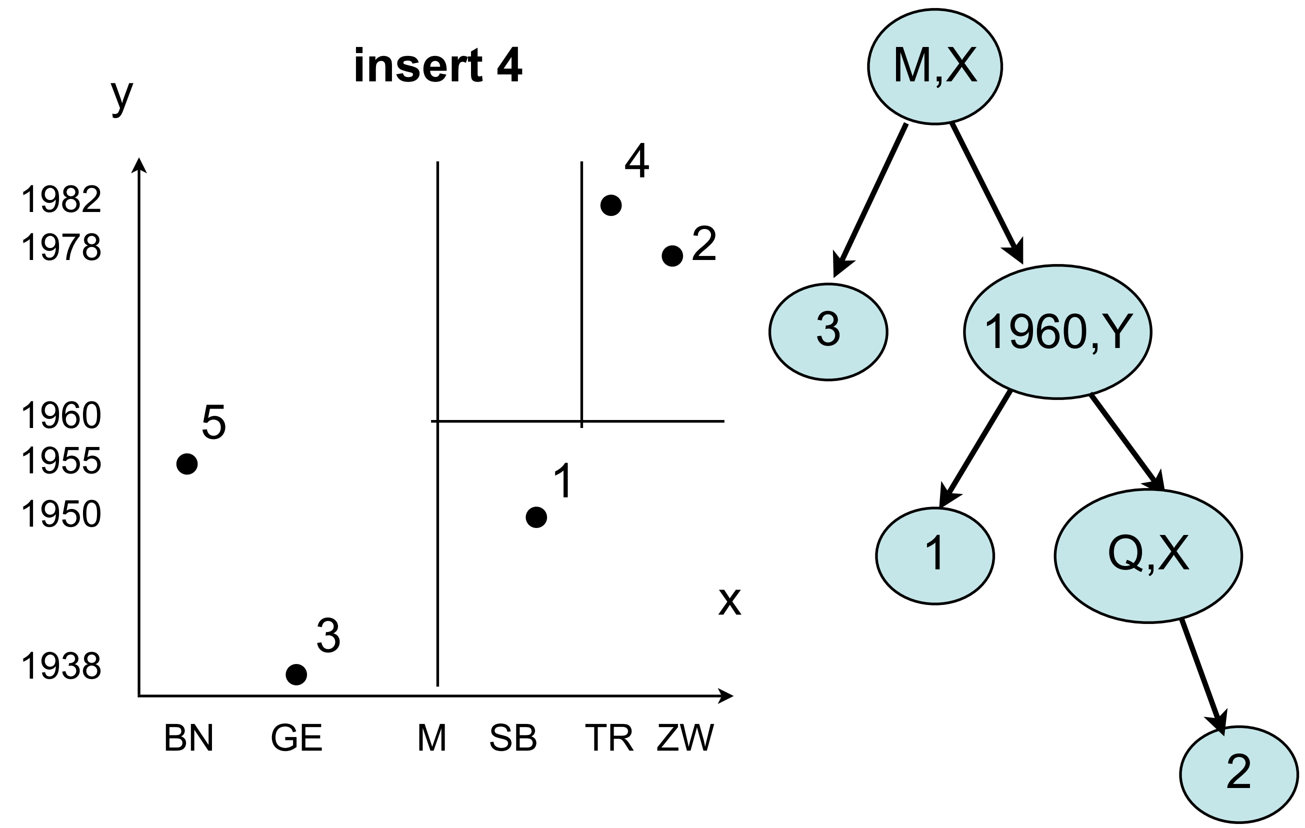
insert 2



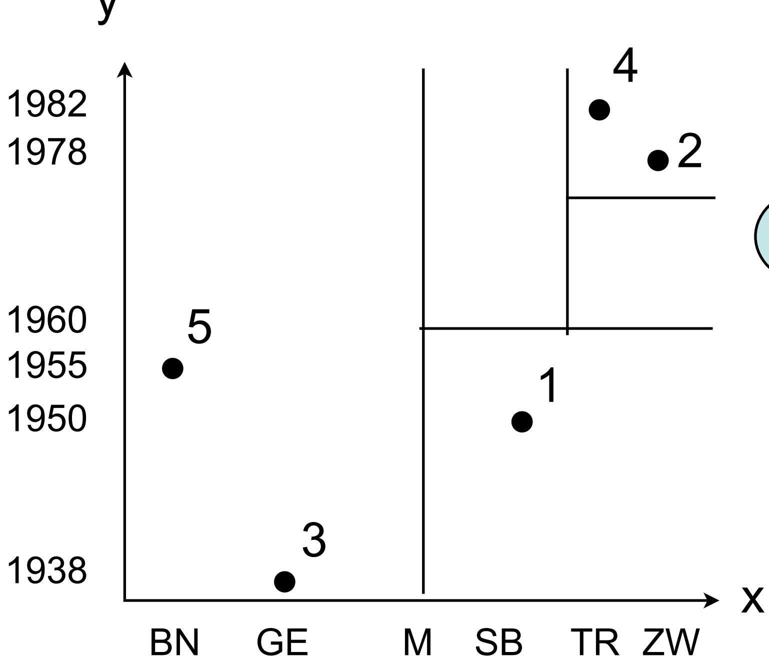
insert 3

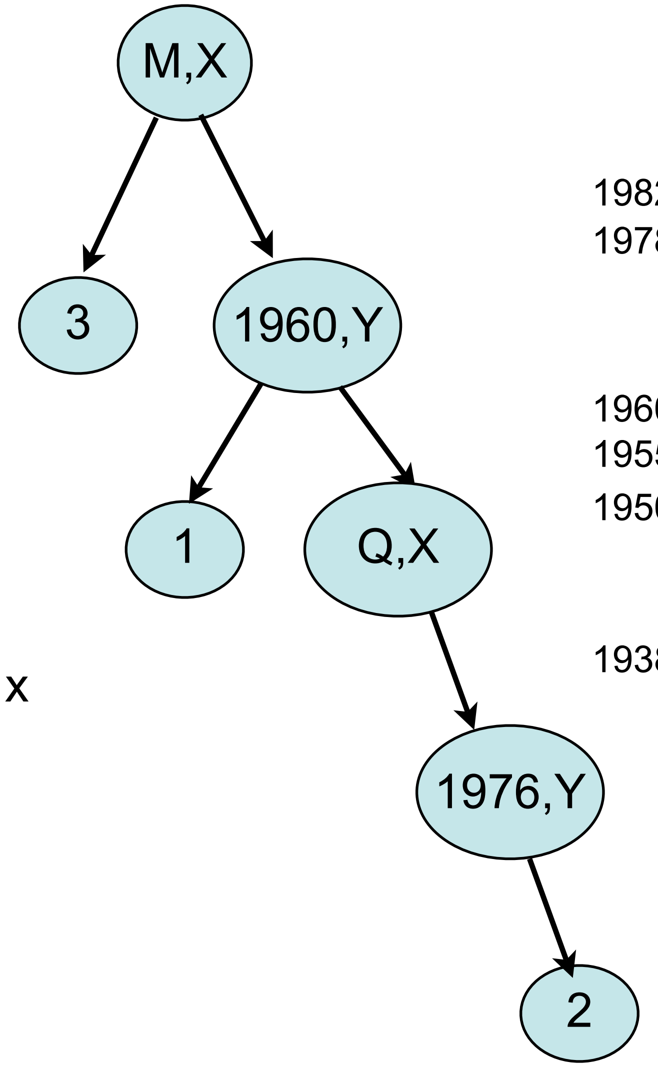


insert 4

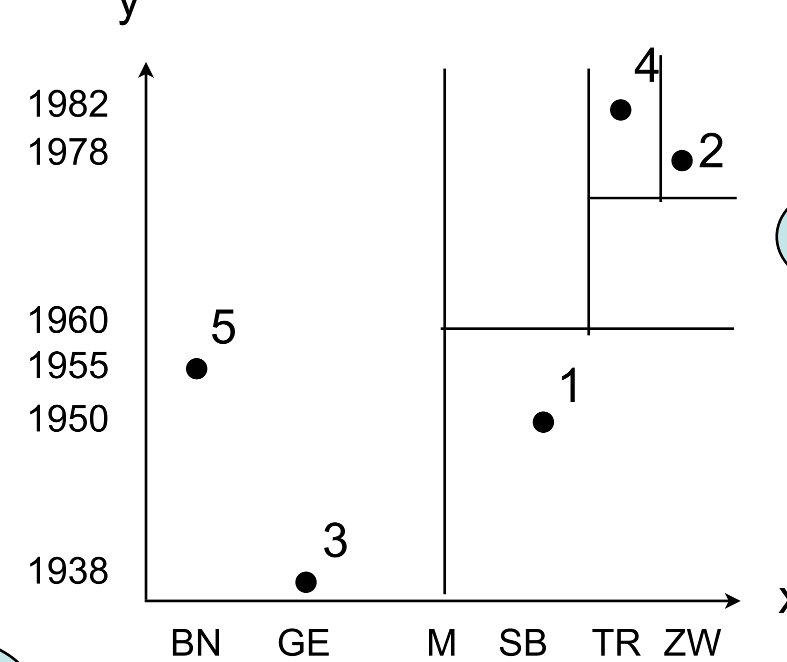


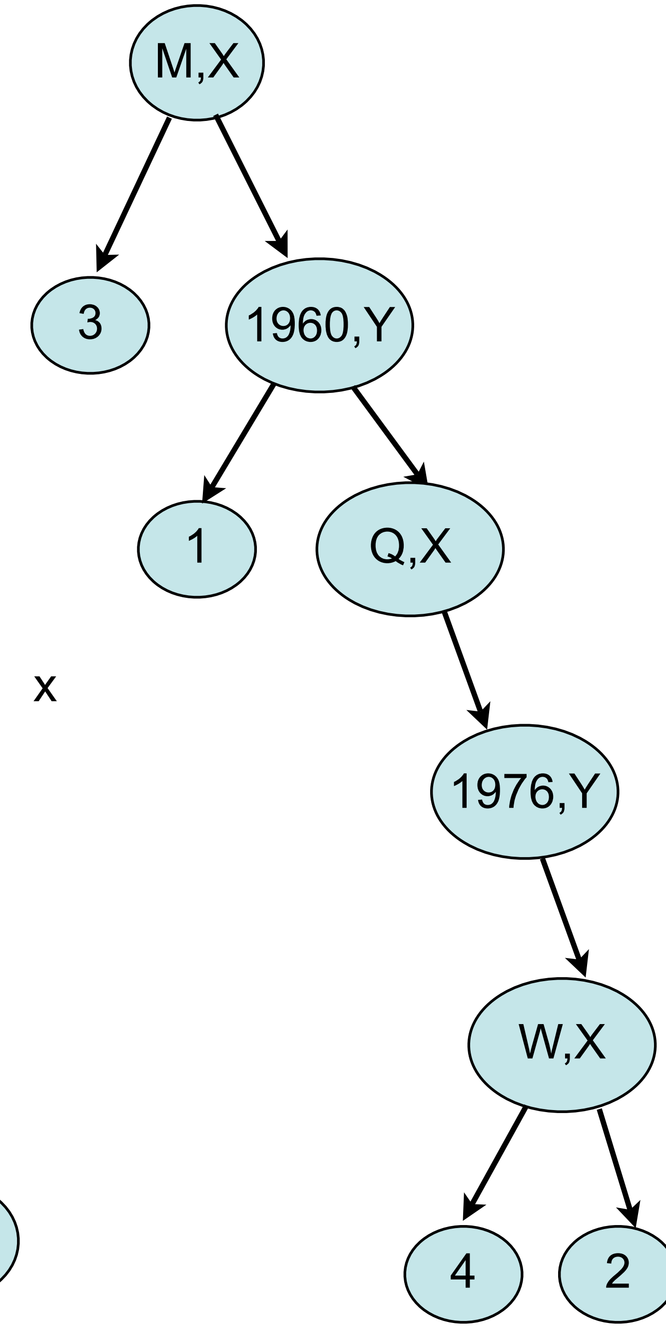
insert 4 (continued)



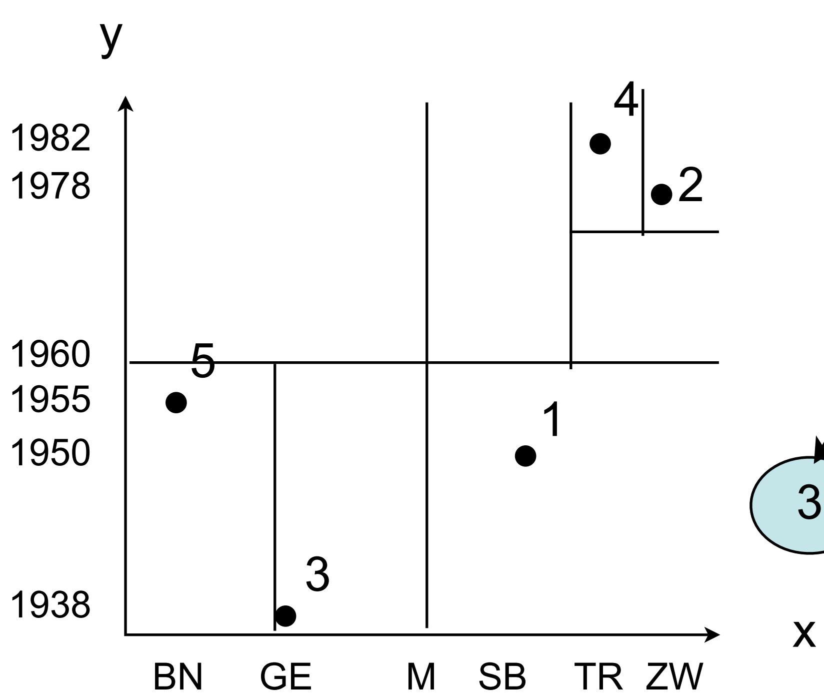


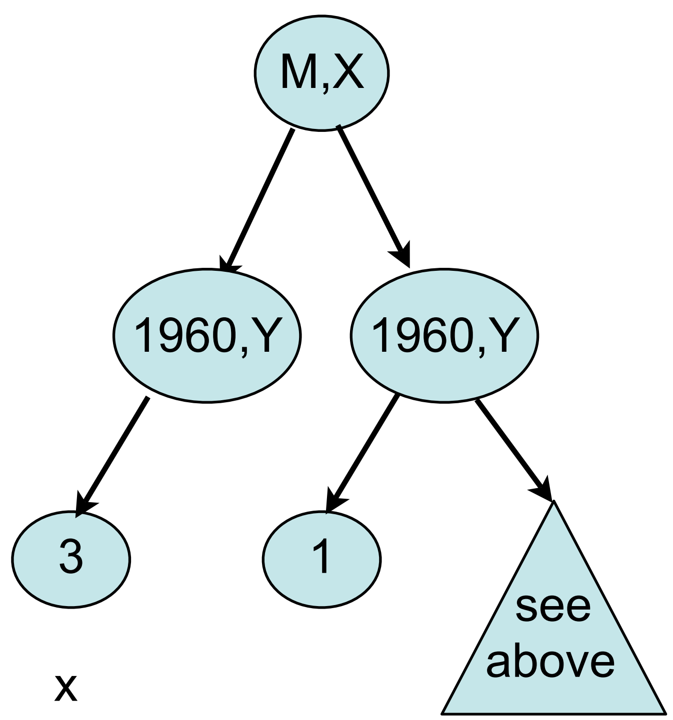
insert 4 (continued)

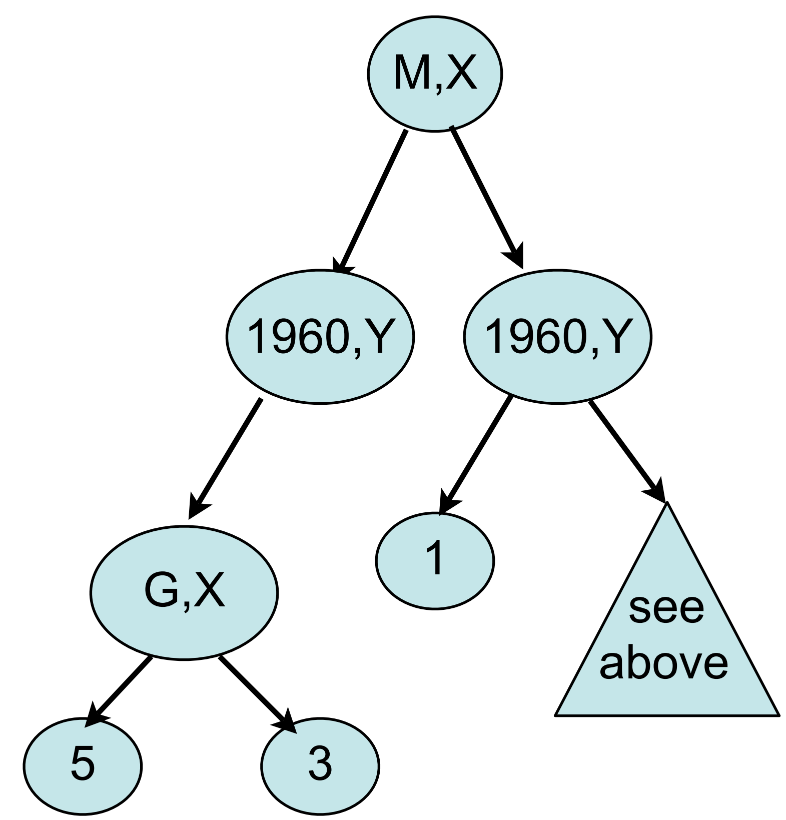




insert 5

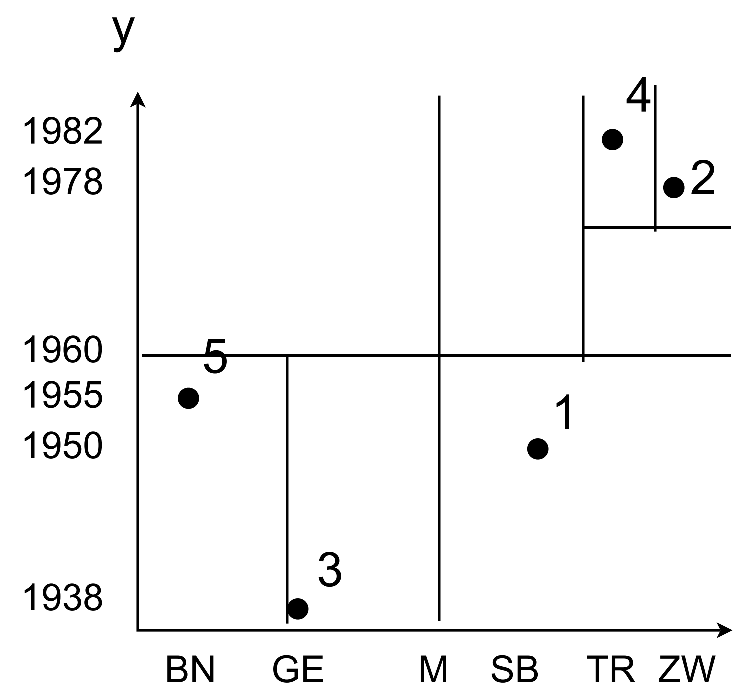


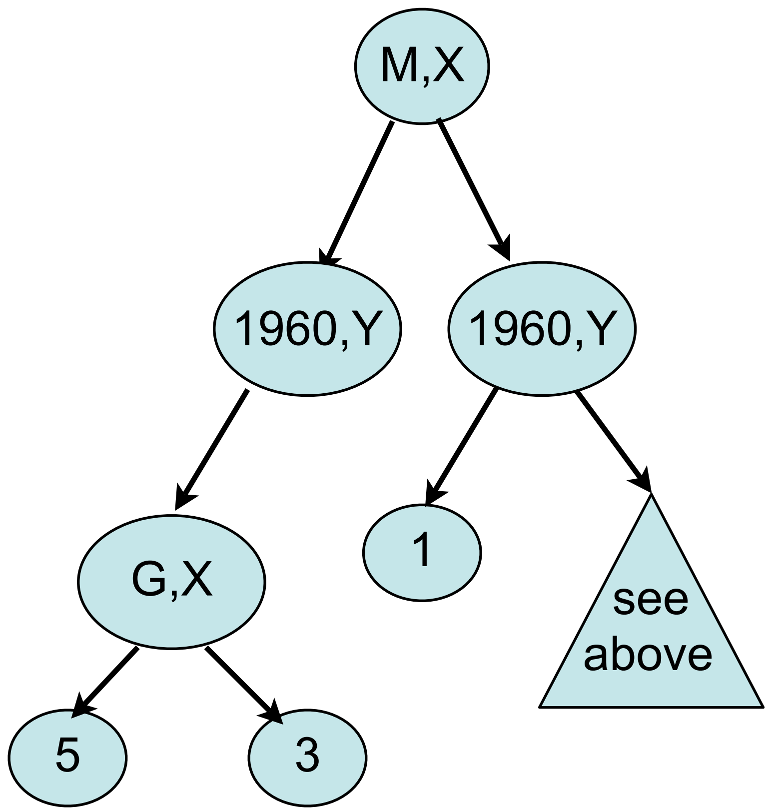




Discussion

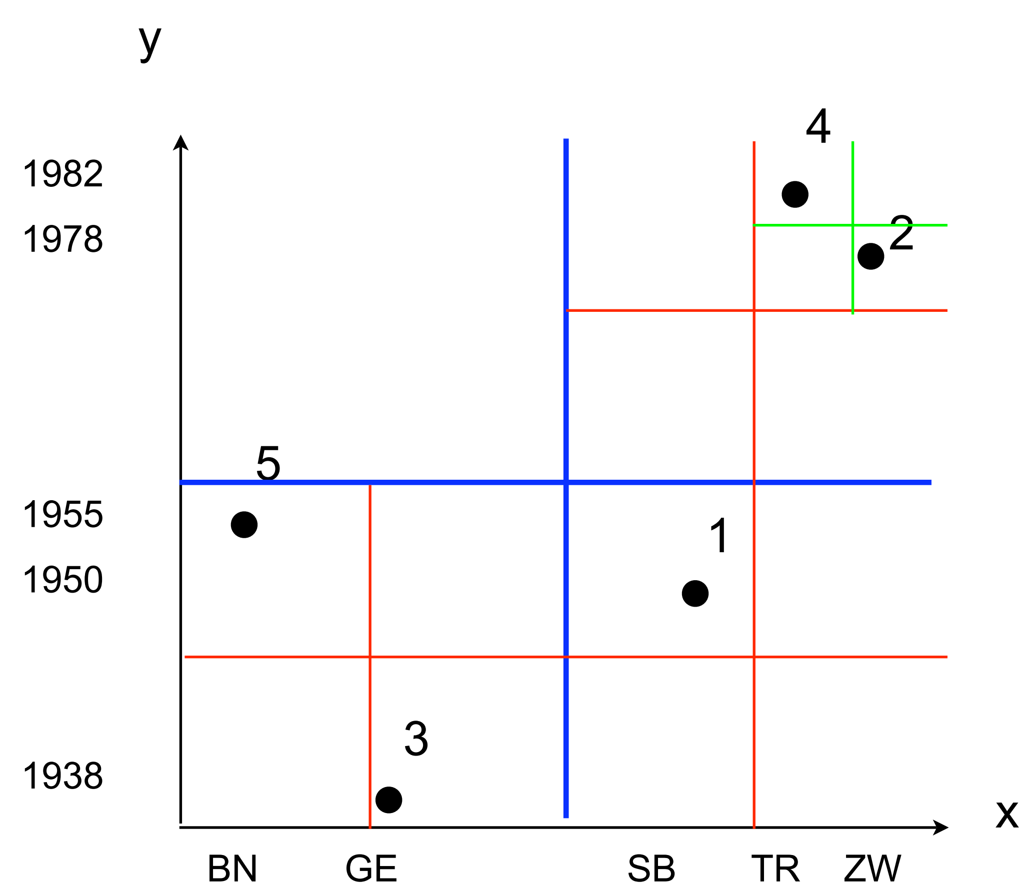
- neither data nor order of insertions impacts partitioning function of the data space
- tree not balanced
- skewed data distributions may degenerate tree into a list
- the partitioning of the data space is **independent from the data**





Quadtree and Quadtrie

- like kd-tree/trie
- But: all dimensions partitioned simultaneously
- every node has a fan-out of 2^d.
- many different variants in literature
- Literature
  - Hanan Samet: The Design and Analysis of Spatial Data Structures. Addison-Wesley. 1990.
  - Hanan Samet: Applications of Spatial Data Structures: Computer Graphics, Image Processing and Gis. Addison-Wesley. 1989.



The story so far

- Observations
  - kd-tries, kd-trees are extensions of binary trees
  - quad-tries and quad-trees are extensions of m-way trees ($m= 2^d$)
- **Question:**
  how do we integrate these structures into a page-oriented DBMS?

kdB-Tree

- map groups of kd-tree nodes to a page, where a page corresponds to a page in a B^+-tree
- a single node partitions the space into a set of disjoint rectangles
- partitioning is complete and disjoint
- split-operations becomes more complicated, underlying B^+-tree-methods have to be adjusted
- Literature:
  - kdB-tree: John T. Robinson: The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes. SIGMOD Conference 1981
  - hB-tree: David B. Lomet, Betty Salzberg: The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. ACM Trans. Database Syst. 15(4): (1990)
  - LSD-tree: Andreas Henrich, Hans-Werner Six, Peter Widmayer: The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects. VLDB 1989.

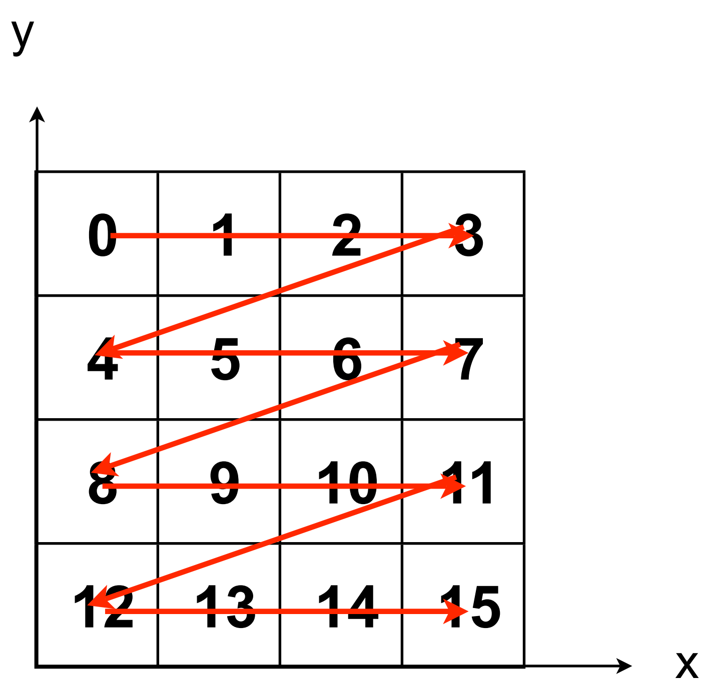
Multi-dimensional Indexes.

Linearized Trees.

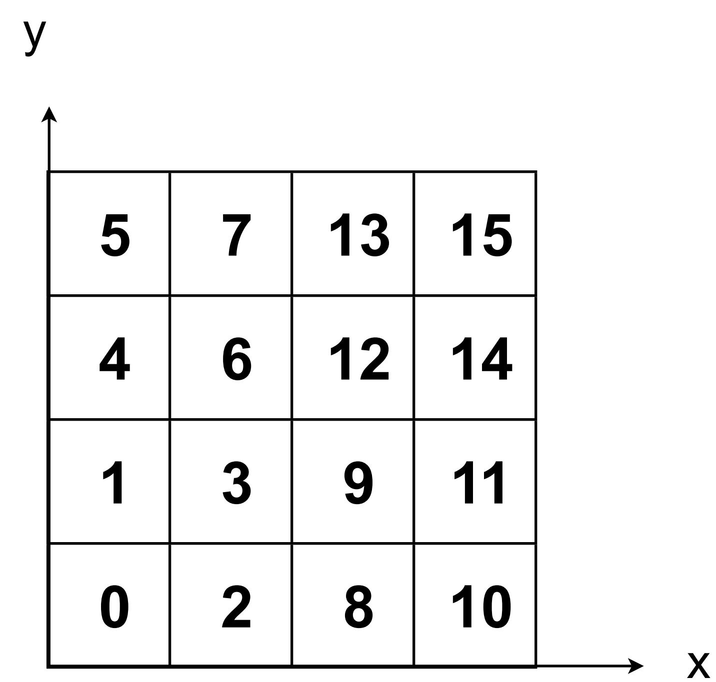
Linearization and B+-trees

- Core idea:
  - number data in a way such that the local clustering of records is preserved
  - number of a record corresponds to the number of a partition
  - use partition numbers as key in a one-dimensional tree structure, e.g., a B+-tree
- Literature
  - H. Tropf and H. Herzog. Multimensional Range Search in Dynamically Balanced Trees. Ang. Informatik, 23(2):71–77, 1981.
  - J. A. Orenstein and T. H. Merrett. A Class of Data Structures for Associative Searching. In PODS, 1984.

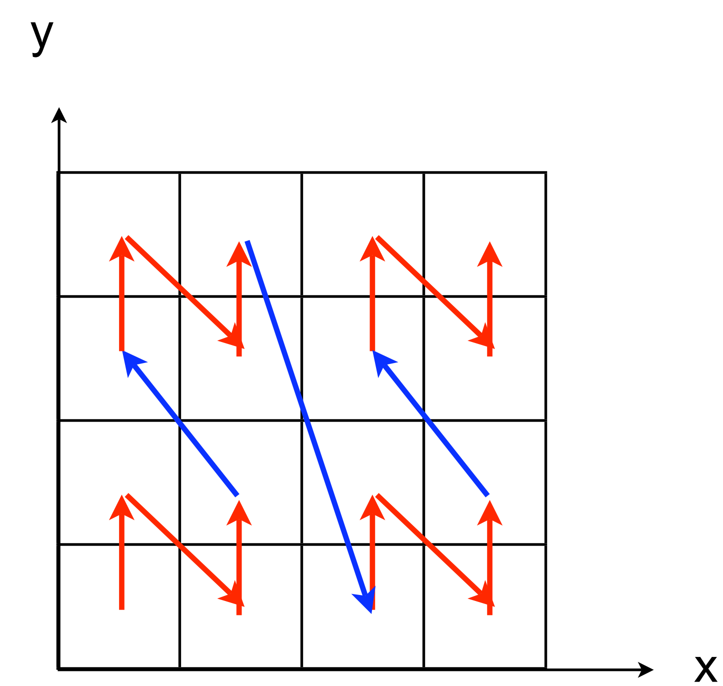
Linearization/Locational Codes



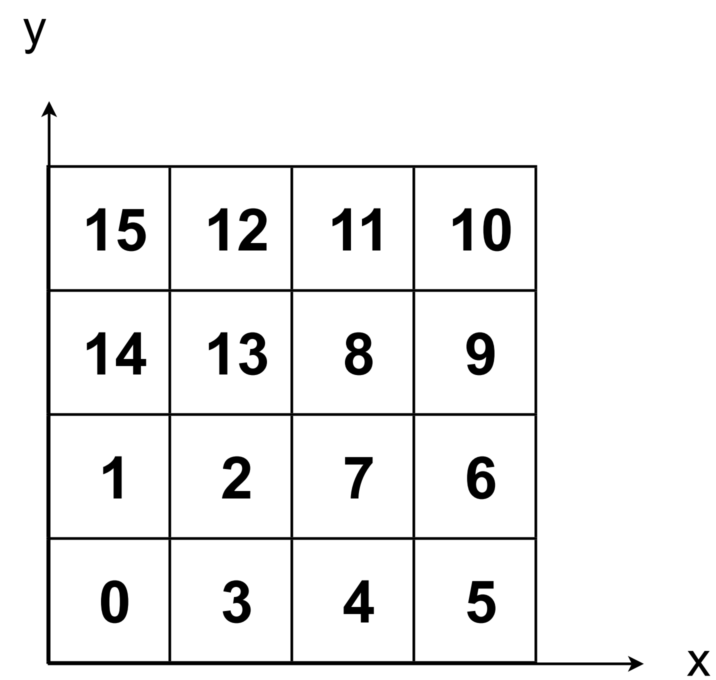
row-wise partition numbering



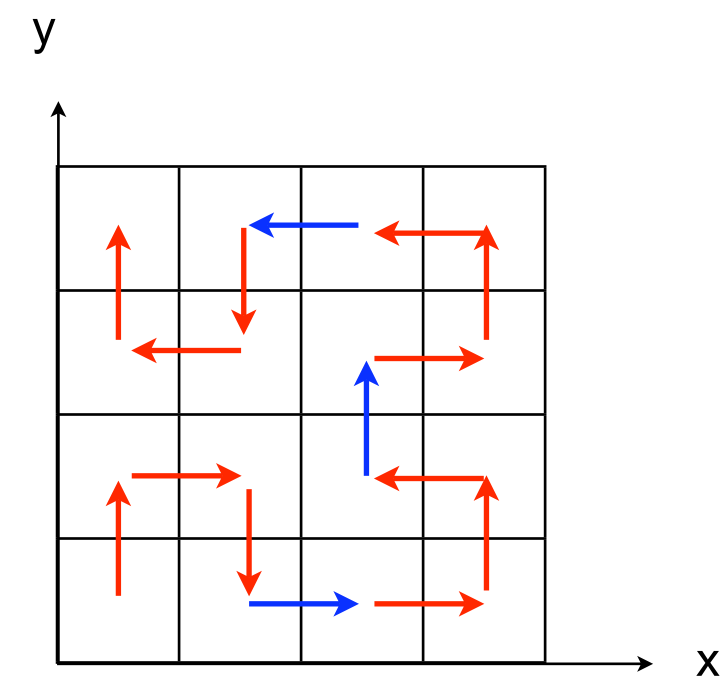
z-order



z-curve

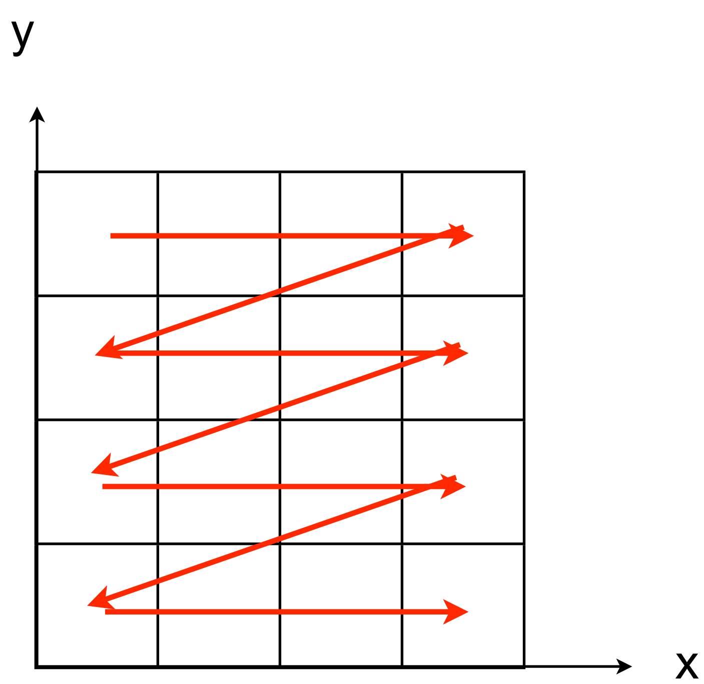


hilbert-order

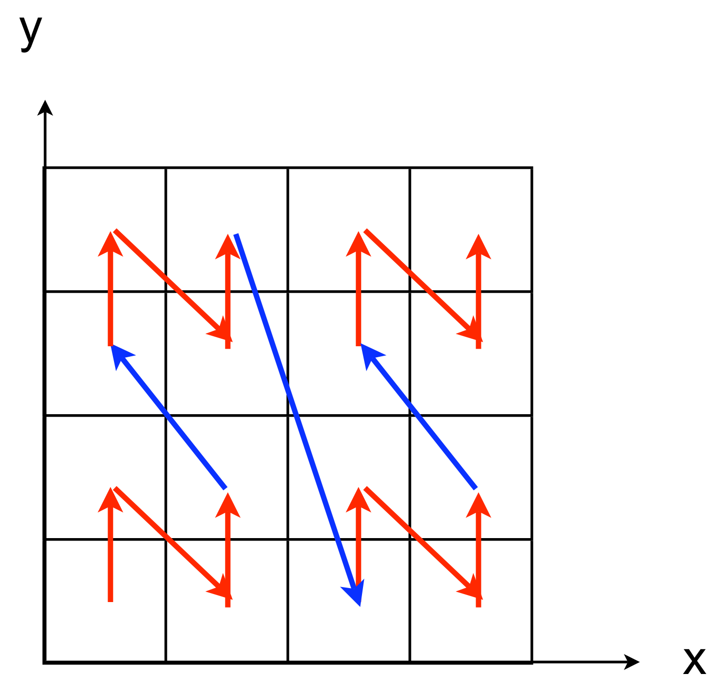


hilbert-curve

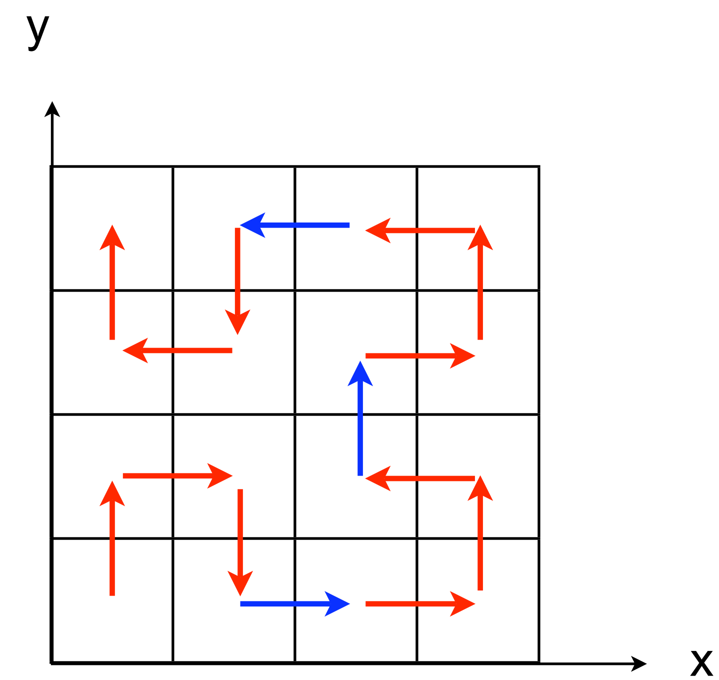
Linearization/Locational Codes



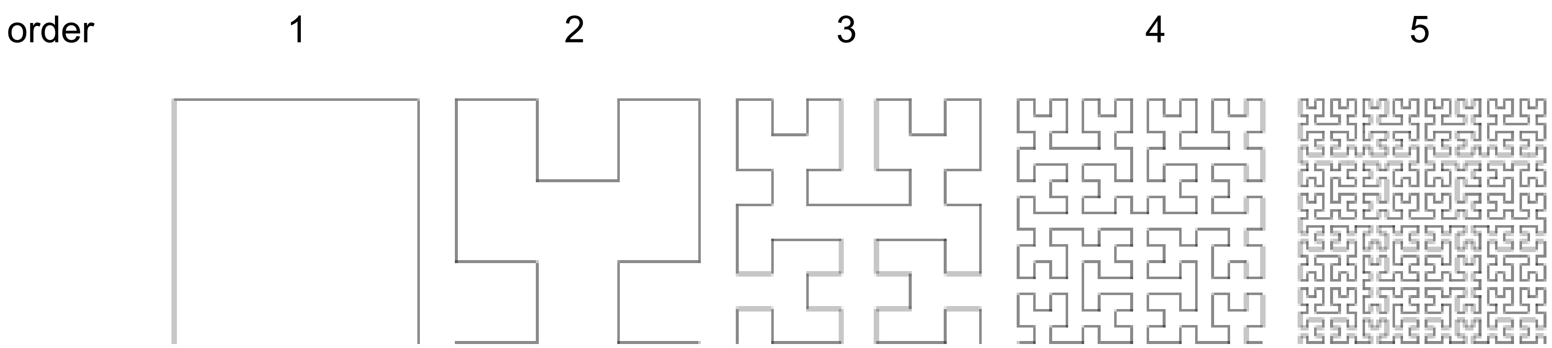
row-wise partition numbering



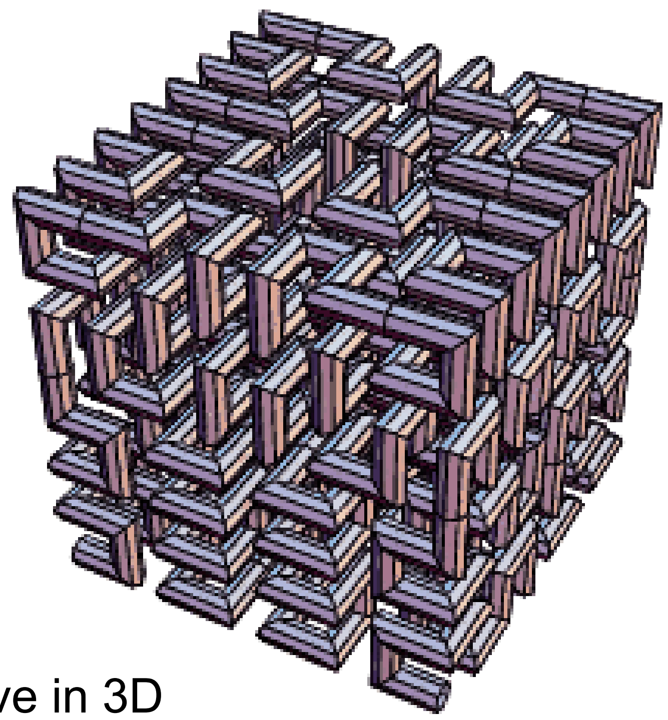
z-curve



hilbert-curve



hilbert-curve in 2D



hilbert-curve in 3D

z-Code Computation

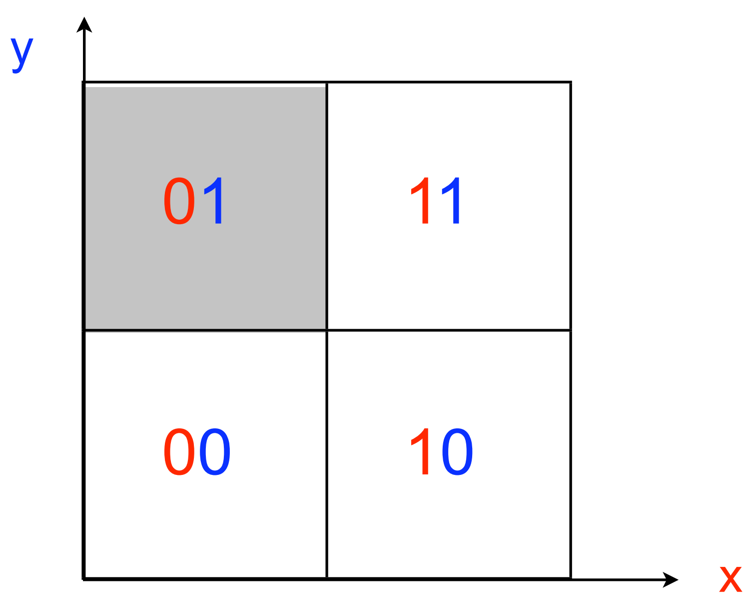
- given a d-dimensional point $v=(v_1,v_2,...,v_d)$
- v_i may be represented as a bit-string $<b_1,...,b_t>$ where $b_k = 0 \mid 1$
- Then, the z-code of order w=t is defined as:

$$
\begin{aligned}
zcode(v) = <&v_1.b_1, v_2.b_1, \ldots, v_d.b_1,\\
&v_1.b_2, \ldots, \qquad v_d.b_2,\\
&\qquad\quad \ldots\\
&v_1.b_w, \ldots, \qquad v_d.b_w>
\end{aligned}
$$

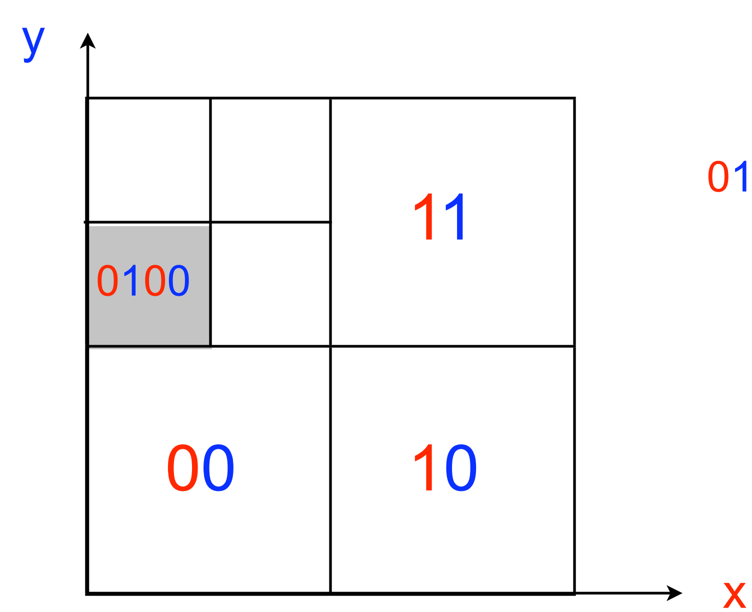
- Example: (d=t=w=3)
  - v=(<001>,<100>,<110>)
  - zcode(v) = <011001100>
- In other words: the bits of the dimensions are “zipped“ together like the teeth of a zip fastener.

z-Code of Order w

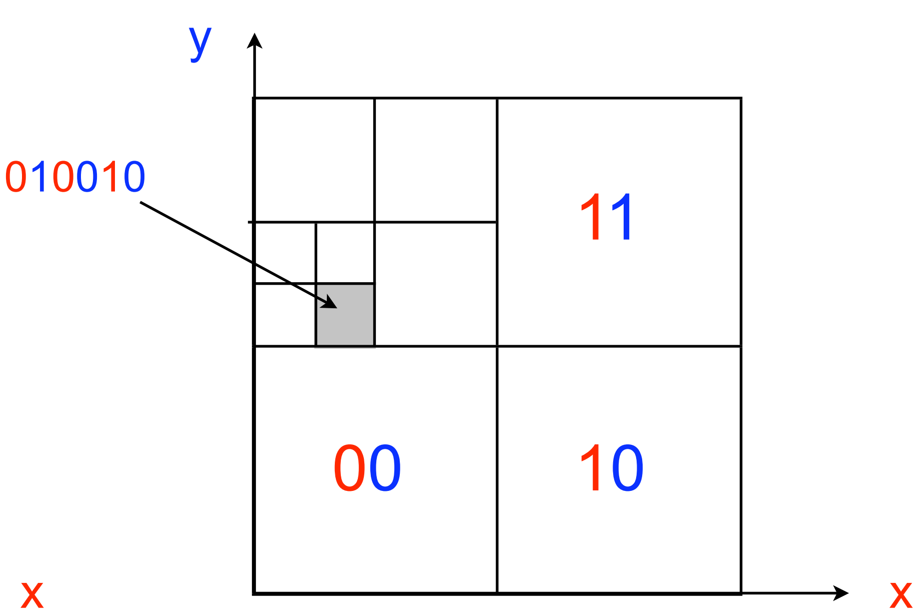
(d=2, t=3), v=(<001>,<100>)



(w=1)
zcode(v) = <01>



(w=2)
zcode(v) = <0100>

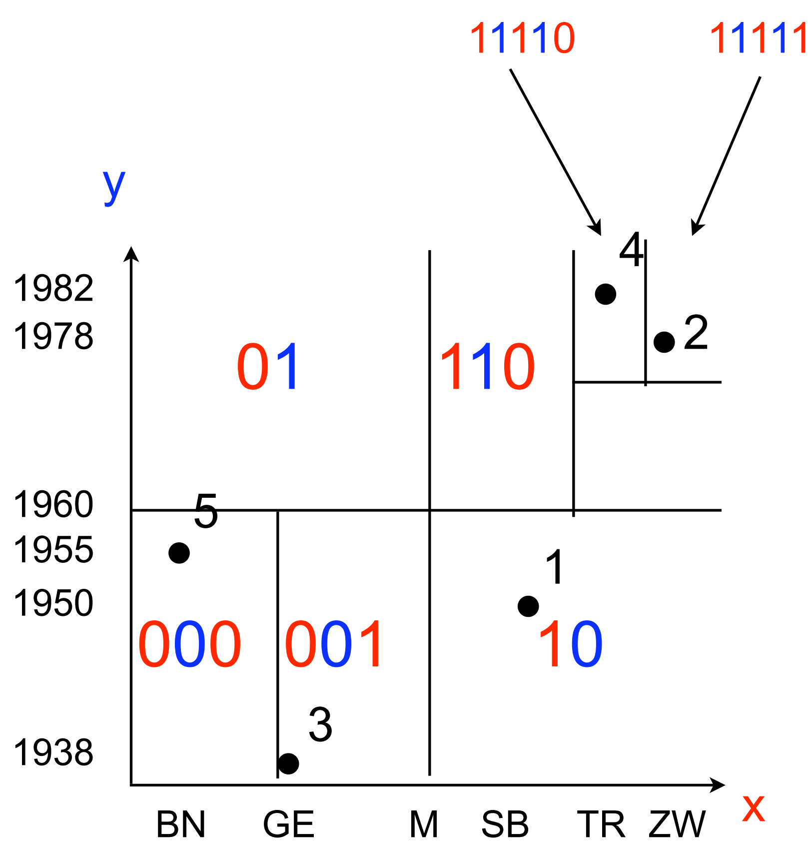


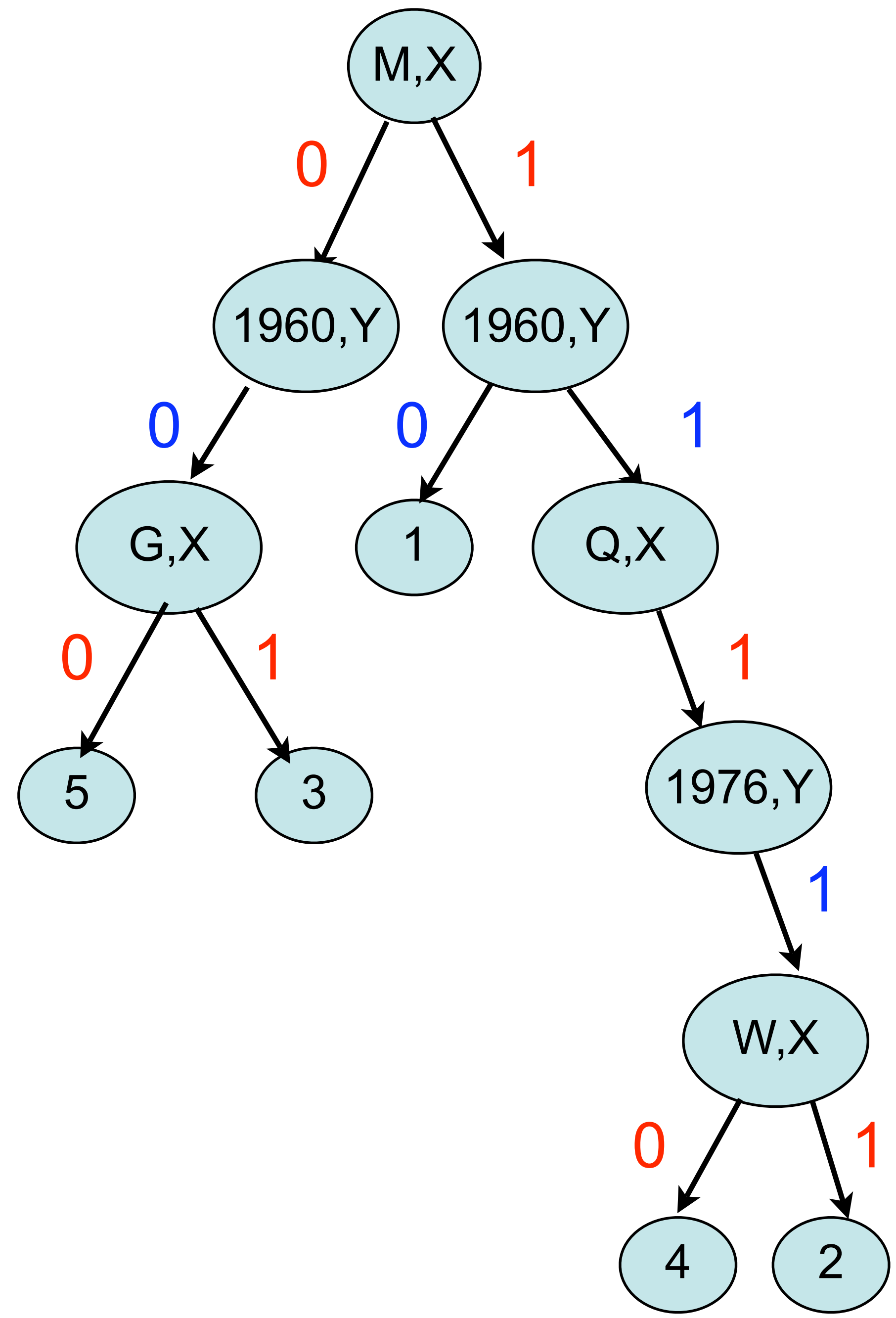
(w=3)
zcode(v) = <010010>

It holds: $zcode(v)_{w1} \leq zcode(v)_{w2}$ for $w1 \leq w2$

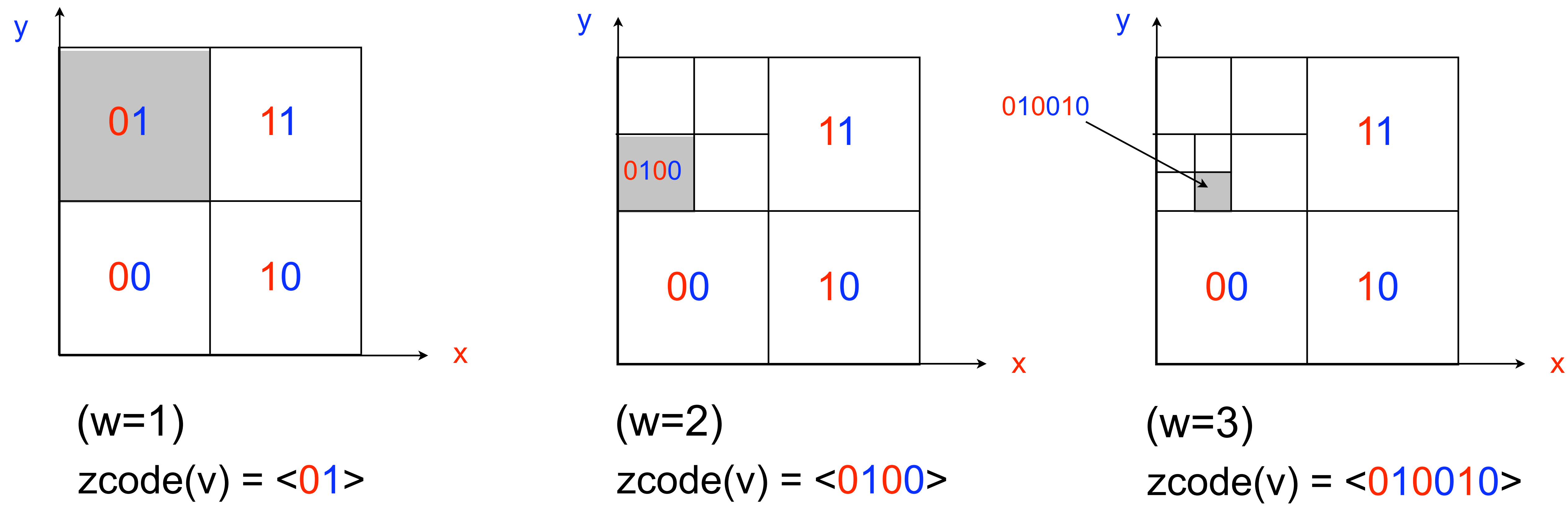
z-Codes and kd-Trie

- Observation: z-code describes the path in a kd-trie:
  - 0: turn left
  - 1: turn right
- analogy: huffman codes





Properties of z-Codes



- It holds:
  - length(zcode(v)) = $d * w$
  - $zcode(v)_{w1} < zcode(v)_{w2}$ for $w1 < w2$
  - $Partition(zcode(v)_{w1}) \supset Partition(zcode(v)_{w2})$ for $w1 < w2$
  - $Prefix(zcode(v)_{w1}, len) = Prefix(zcode(v)_{w2}, len)$, $len = d * min(w1,w2)$

Linearization and B+-trees: insert (1/2)

key

value

- **instead of**:

  bplustree.insert (ZW1982, TID);

- **now:**

  bplustree.insert (zcode(ZW, 1982), TID);

- **Note**
  - mapping may generate duplicate keys!

Linearization and B+-trees: insert (2/2)

- **point query**

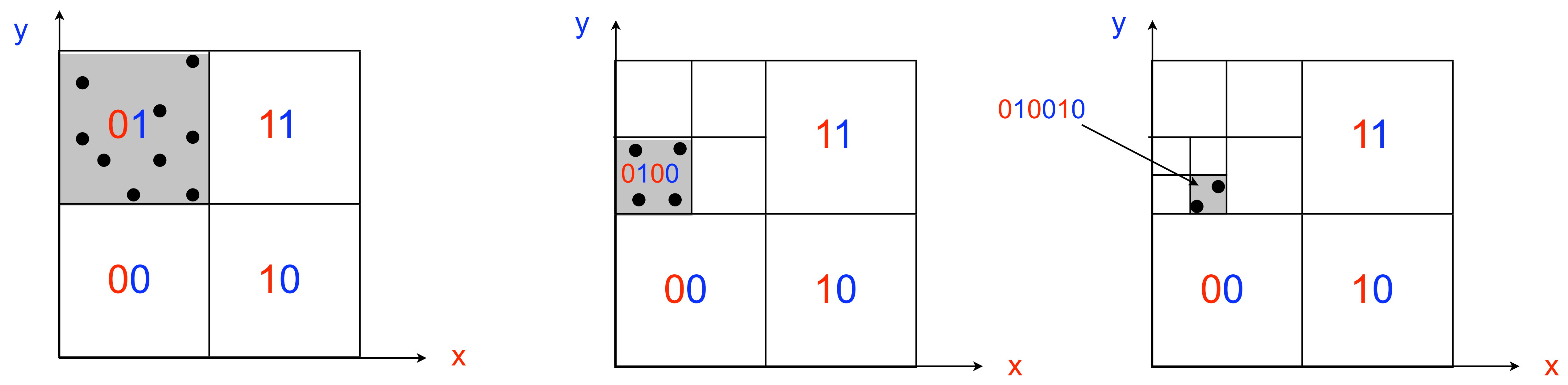
  candidateSet = bplustree.find_key (zcode(ZW, 1982));

- **Note**
  - candidateSet is a **superset** of the desired result!
  - This means candidateSet has to be postfiltered:

    result = { v | v in candidateSet
    and v.city = 'ZW'
    and v.agegroup = 1982
    }
  - Reason: depending on the granularity of the z-codes used different records may map to the same z-code.
  - similar problem as in hashing
  - remember extendible hashing?

Duplicates and Containment

(d=2, t=3), v=(<001>,<100>)



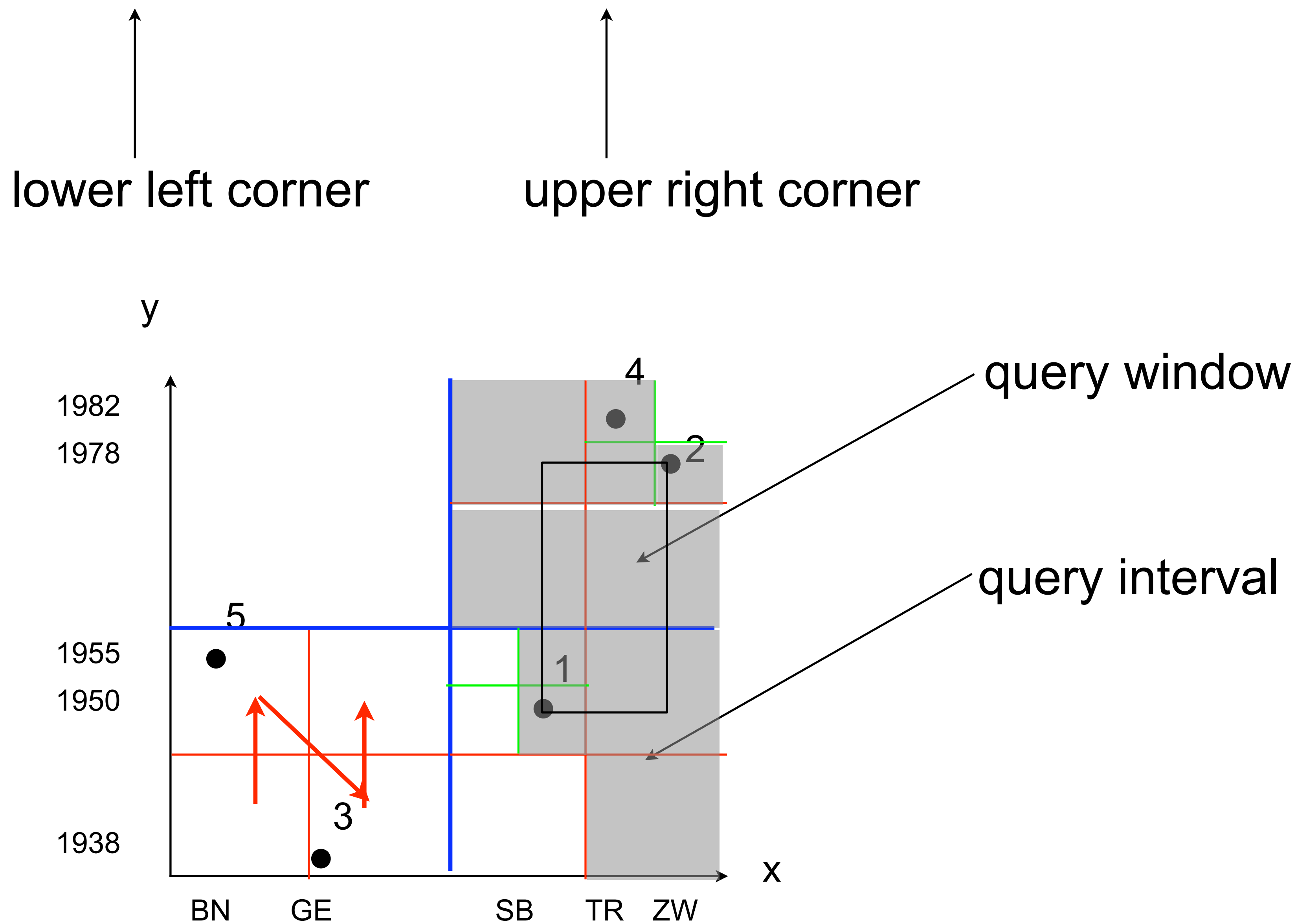
(w=1)
zcode(v) = <01>

(w=2)
zcode(v) = <0100>

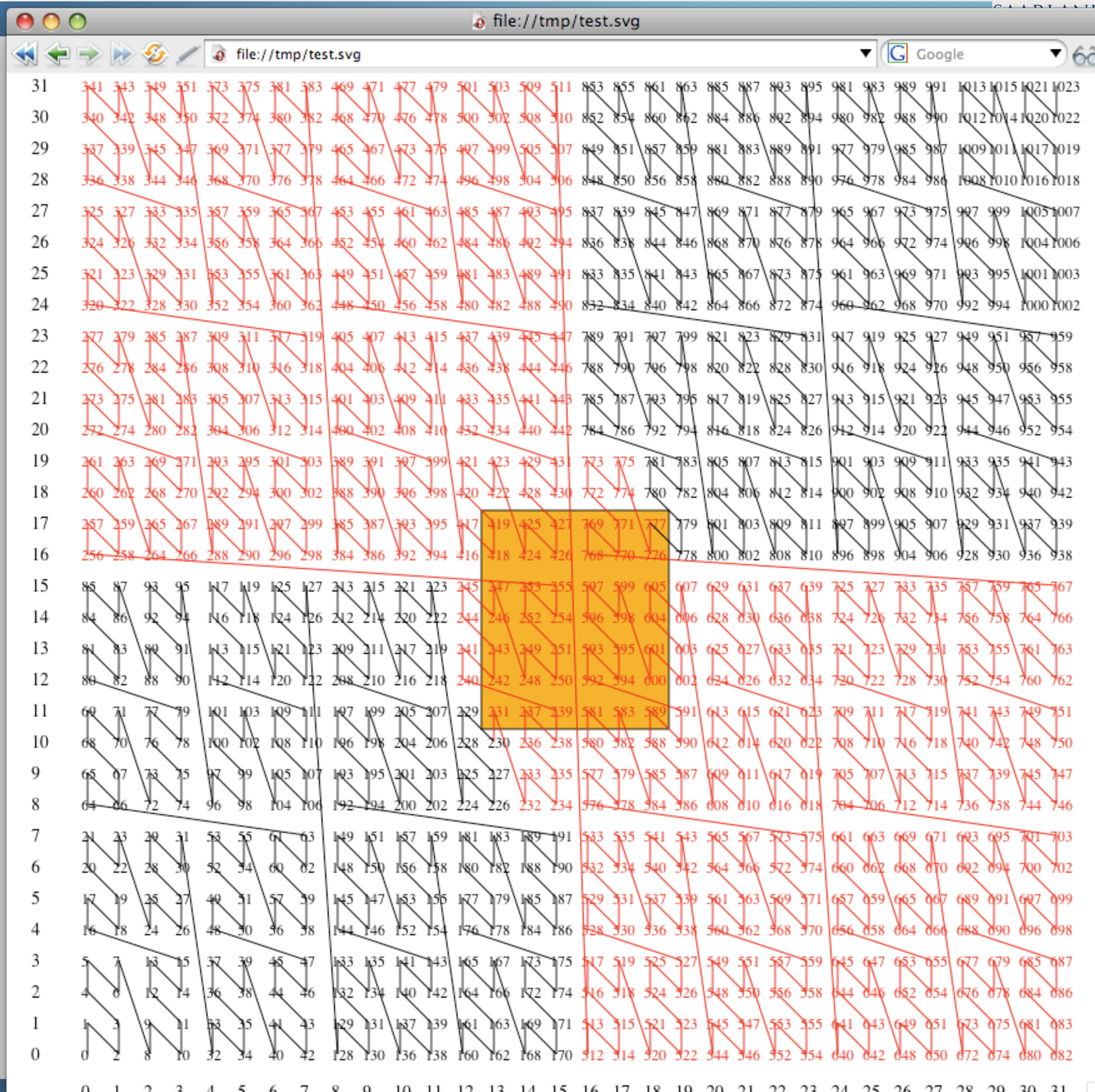
(w=3)
zcode(v) = <010010>

Linearization and find_range

- 1. approach: naïve range query:
  - find_range ([zcode(SB, 1950); zcode(ZW, 1978)])

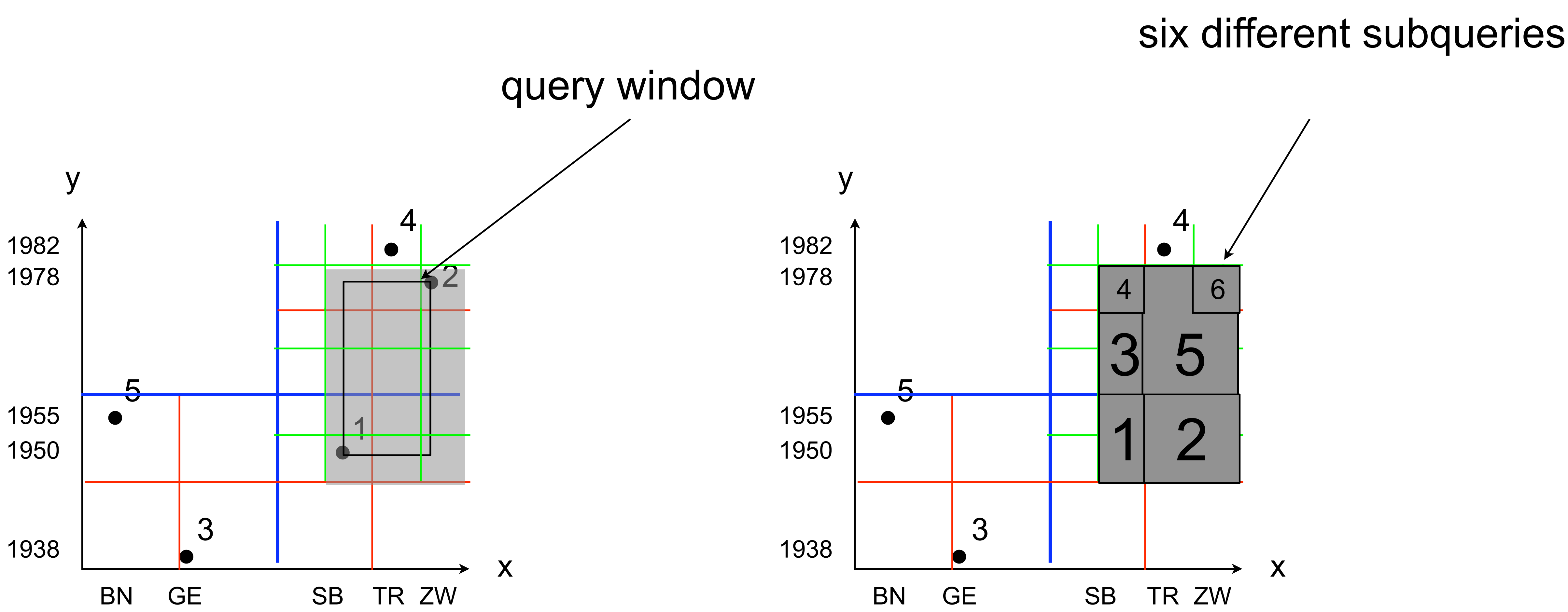


Demo



Linearization and find_range

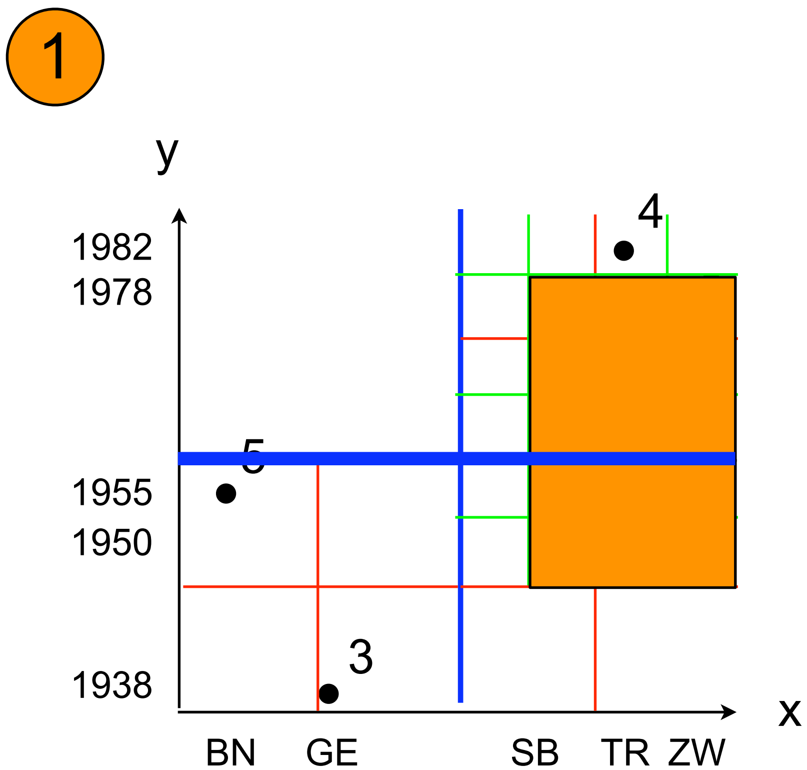
- 2. approach: range query based on partitioning:
  - split original query into subqueries



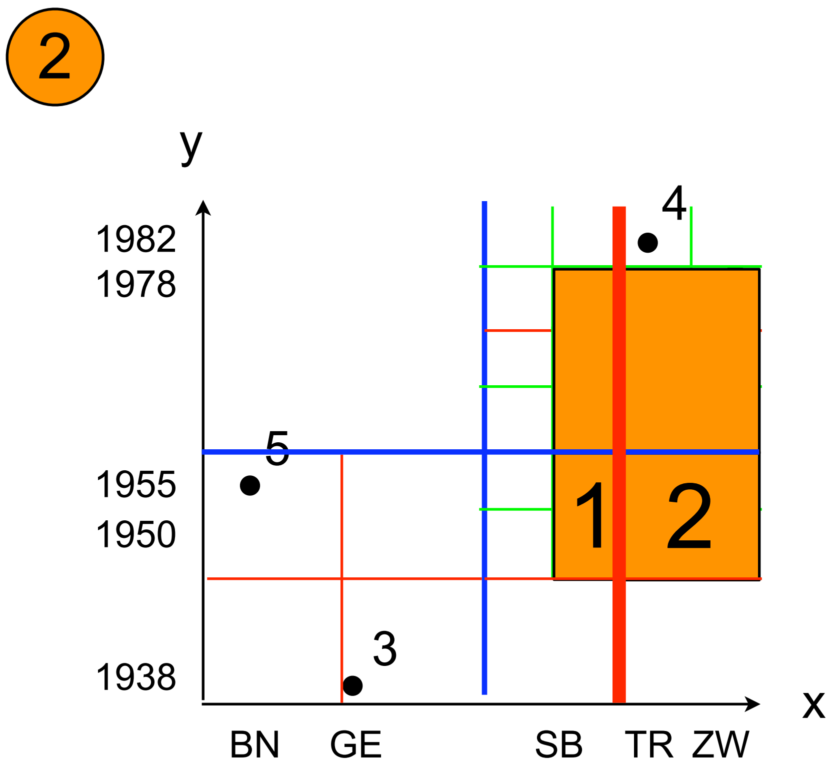
Question: how to compute the query partitioning?

Computing a Query Partitioning (1/2)

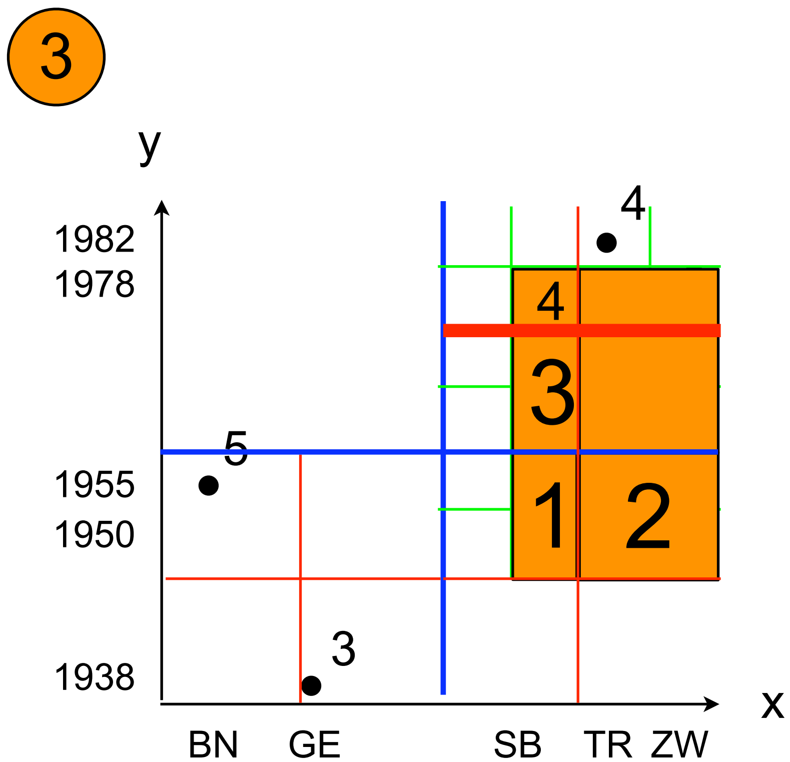
- core idea: split query window recursively along split lines



first split at bit 1
of y-dimension:
2 partitions

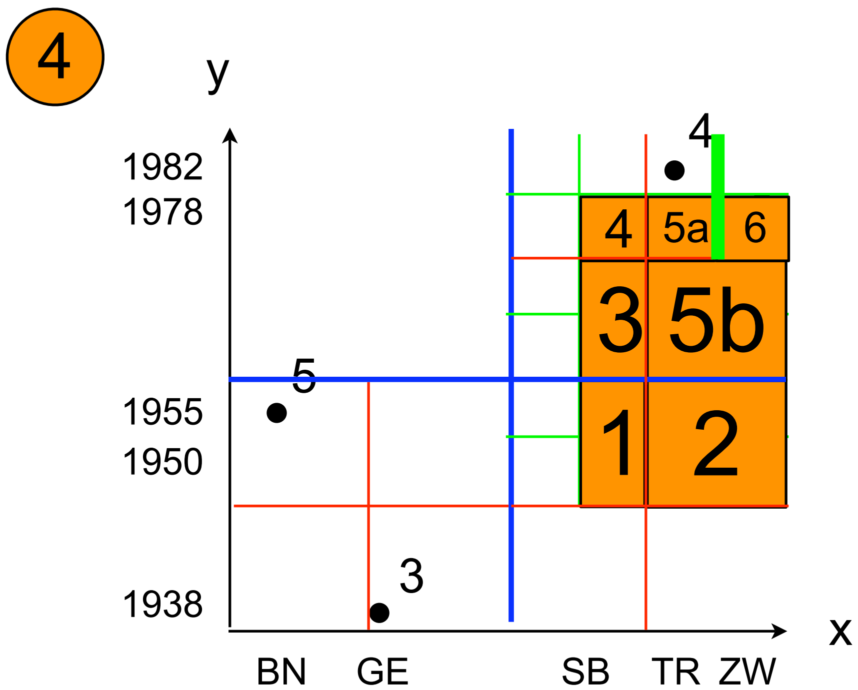


second split at bit 2
of x-dimension:
4 partitions

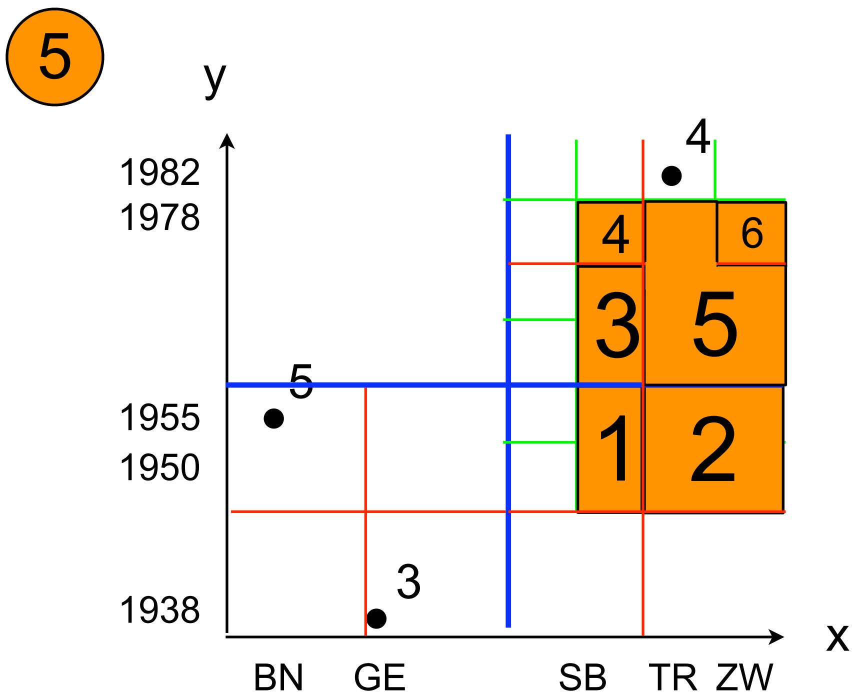


third split at bit 2
of y-dimension:
6 partitions

Computing a Query Partitioning (2/2)



fourth split at bit 3
of x-dimension:
7 partitions



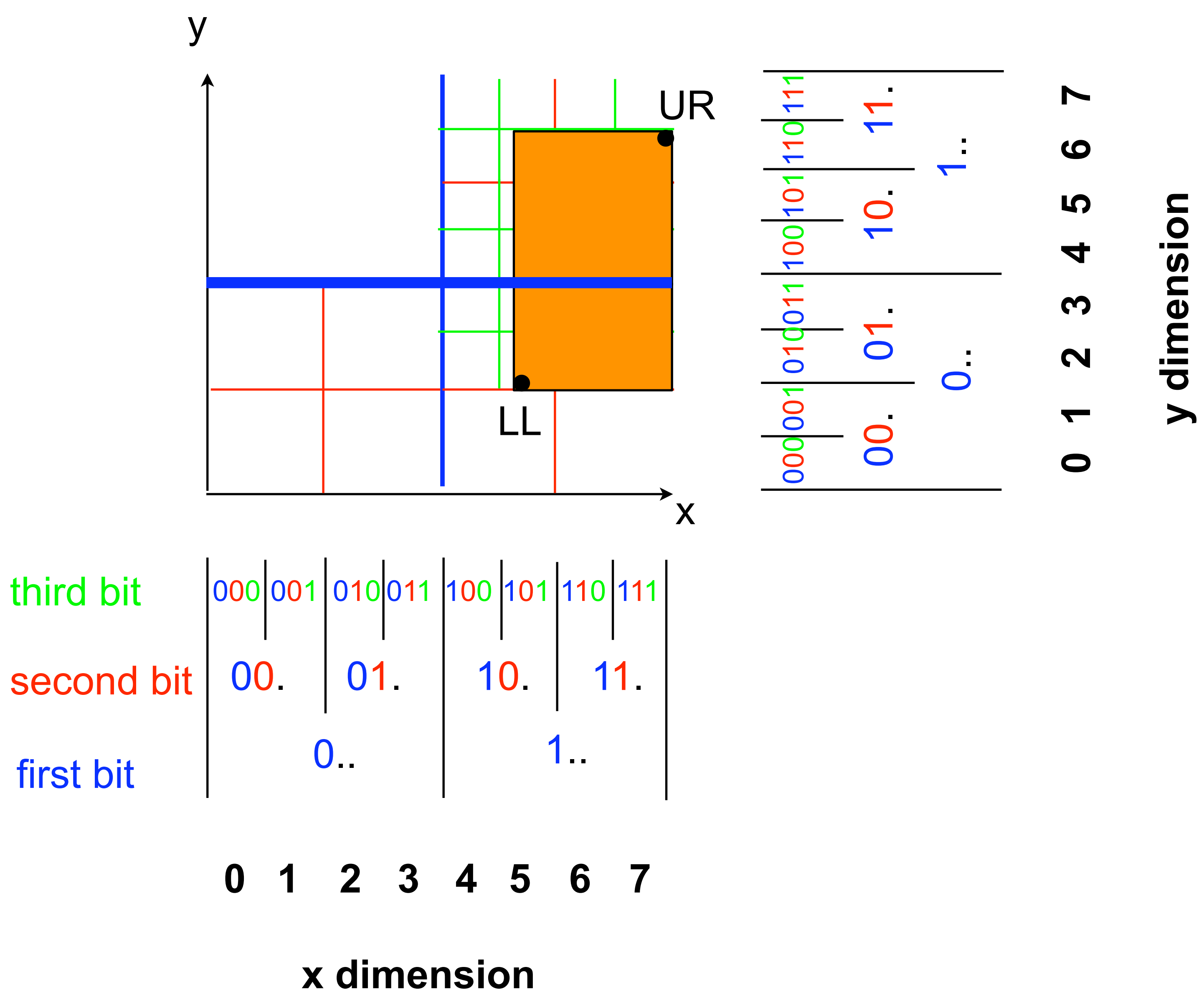
partitions 5a and 5b are
adjacent in z-order
=>merge partitions:
6 partitions in total

- recursion may be continued until perfect partitioning is obtained

The "Dirty Details"

- how to make a single query partitioning efficient
  => bit-operations
- possible if domain can be mapped to integer domain
- bit-representation of integers then corresponds 1:1 to split lines!
- two approaches
  - 1. partition the query window, then recompute z-intervals
    (a bit easier to understand)
  - 2. partition the z-intervals directly
    (magic, however just an inlined variant of approach 1)

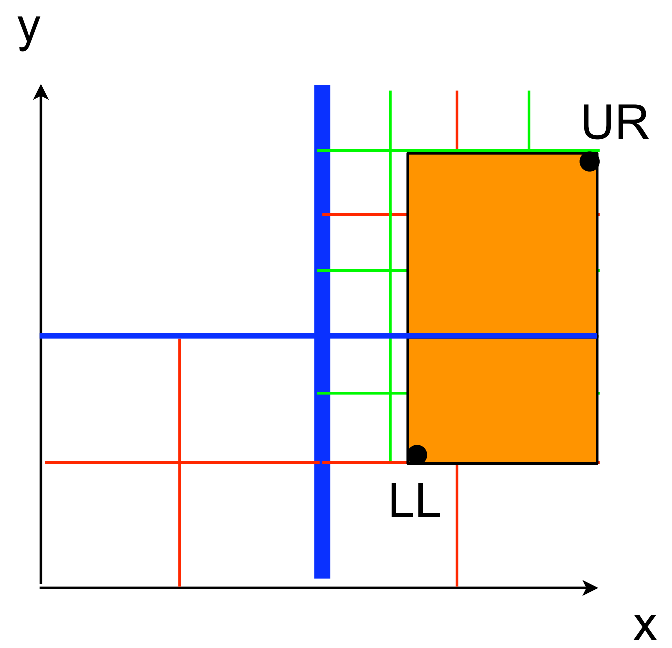
Partition the Query Window



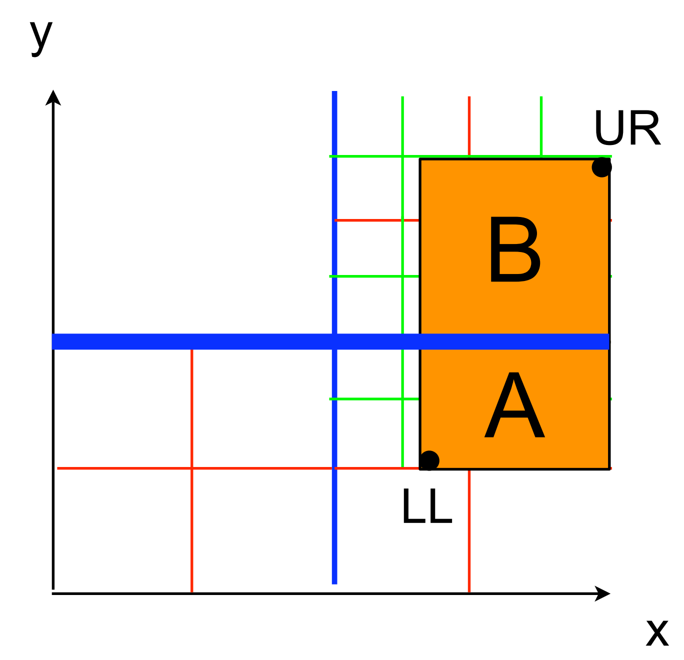
- for each dimension it holds: the binary representation of the integer determines the partition in that dimension
- example: LL
  x=5=101$_b$, y=2= 010$_b$
- UR
  x=7=111$_b$, y=6= 110$_b$
- algorithm:
  - compare bits **dimension-wise**
  - if bits differ, we have hit a split line => split interval

Partitioning Example (1/3)

- example: LL
  x=5=101$_b$, y=2= 010$_b$
- UR
  x=7=111$_b$, y=6= 110$_b$
- bits equal
- => we did not hit the split line
- => no split happens!

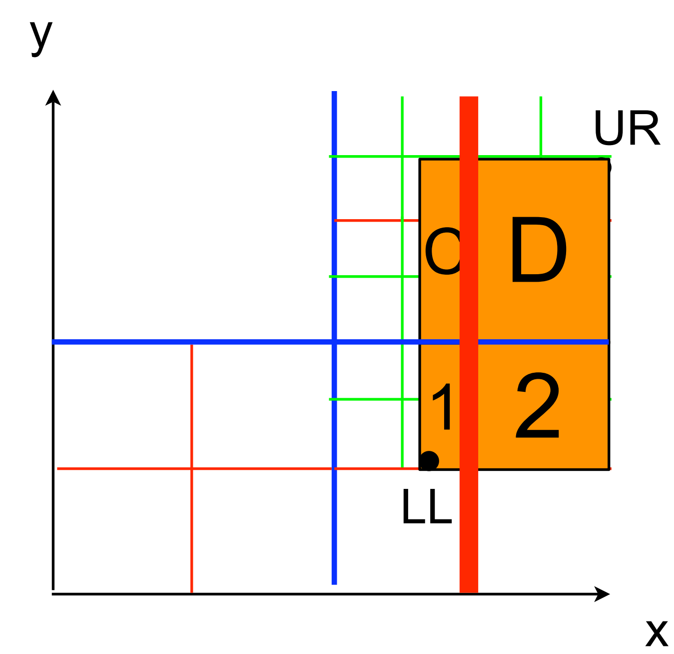


Partitioning Example (2/3)



- example: LL
  x=5=101$_b$, y=2=010$_b$
- UR
  x=7=111$_b$, y=6=110$_b$
- bits differ
- => we have to split the rectangle along its y-dimension
- we split the y-interval!
- [010;110] => [010;011], [100;110]
- i.e. [2;6] => [2;3], [4;6]
- thus we obtain two rectangles
  A=[5;7] x [2;3] => z-interval(A) = [100110;101111] = [38;47]
  B=[5;7] x [4;6] => z-interval(B) = [110010;111110] = [50;62]

Partitioning Example (3/3)



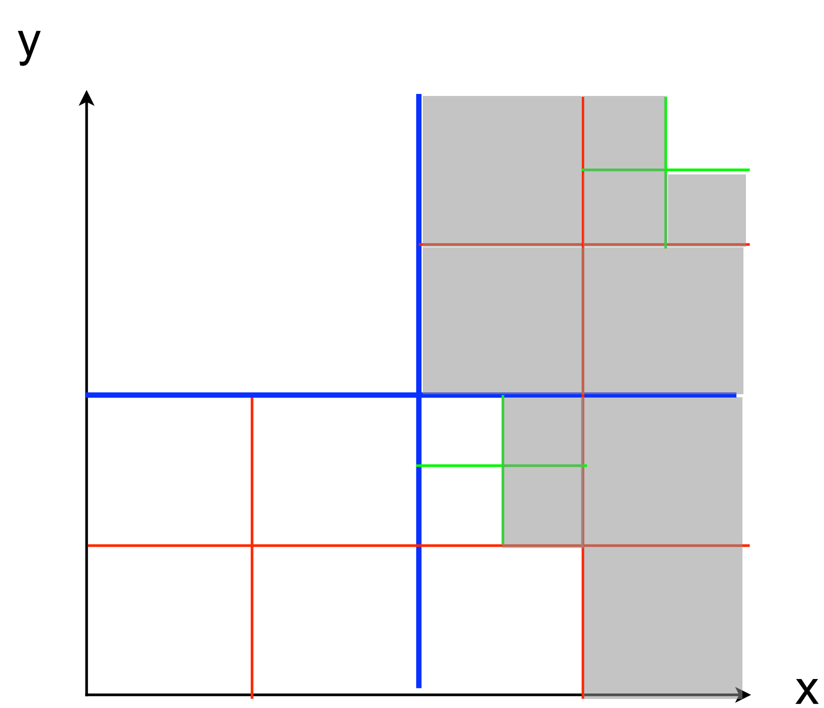
- example: LL
  x=5=101$_b$, y=2= 010$_b$
- UR
  x=7=111$_b$, y=6= 110$_b$
- bits differ
- => we have to split both rectangles A,B along their x-dim
- we split the x-interval!
- [101;111] => [101;101], [110;111]
- i.e. [5;7] => [5;5], [6;7]
- thus we obtain four rectangles 1=[5;5]x[2;3], 2=[6;7]x[2;3], C=[5;5]x[4;6], D=[6;7]x[4;6]
  z-int(1)=[100110;100111]=[38;39],
  z-int(2)=[101100;101111]=[44;47]

Splitting Rule

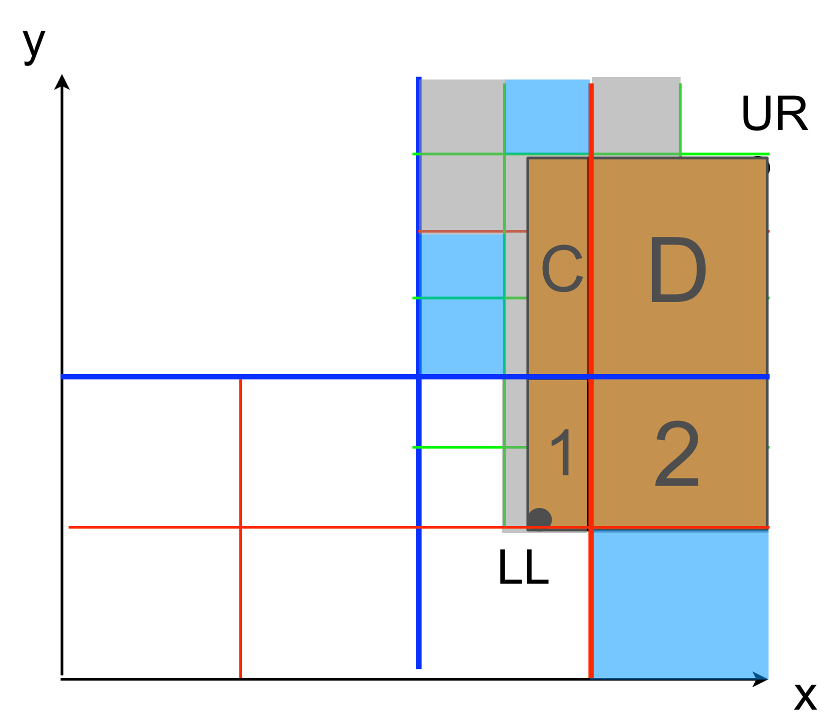
- we check dimensions in round-robin following the kd-trie partitioning scheme
- whenever we compare two numbers
  x_{low} = <$b_{low\ 1}$,...,$b_{low\ t}$> and x_{high} = <$b_{high\ 1}$,...,$b_{high\ t}$>
  at position $1 \leq i \leq t$
- and $b_{low\ i} == b_{high\ i}$
- => we continue, i.e., move to the next split line
- otherwise, i.e. $b_{low\ i} \neq b_{high\ i}$, we split and compute the split intervals:
- [x_{low} ; $\mathbf{x'_{low}}$] , [$\mathbf{x'_{high}}$; x_{high}]
- $\mathbf{x'_{low}}$ = like x_{low}, however bit $b_{low\ i}$=0, $b_{low\ j}$=1 for j>i, pattern ...011111111...
- $\mathbf{x'_{high}}$ = like x_{high}, however bit $b_{high\ i}$=1, $b_{high\ j}$=0 for j>i, pattern ...100000000...

What we gain

- note that the split of an interval may not create any “holes“
- i.e., the partitioning is a perfect disjoint partitioning
- we only gain in the second step when we compute z-intervals for the individual partitions
- that step may cut out intervals from the original z-interval



unpartitioned, naive query



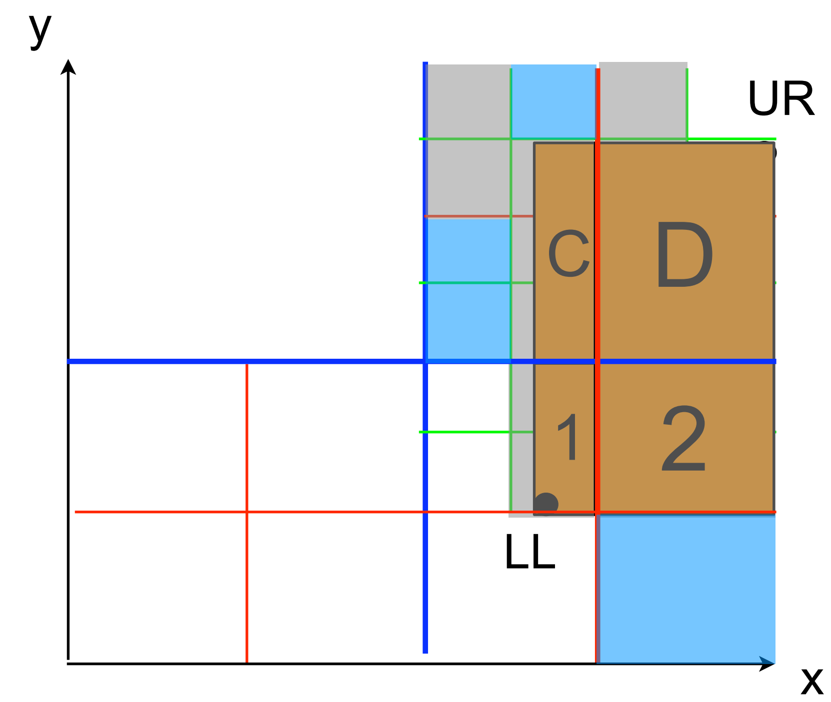
partitioned query

How to Stop Recursion

- we could continue the recursion until no dead space is retrieved anymore
- however this may lead to quite a number of partitions
- as each partition may trigger random I/O we have to find a good trade-off
- one approach: consider ratio

$$\frac{\text{region covered by partitions}}{\text{region covered by the query}}$$

- stop recursion if ratio below a given threshold
- determine threshold => experiment



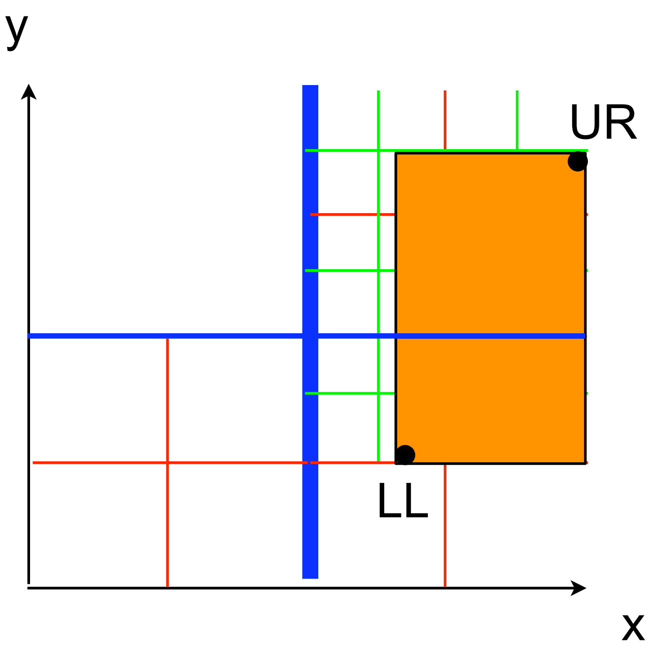
partitioned query

Partition the z-Intervals directly

- basically an inlined version of the previous approach
- advantage: no extra z-code computation
- algorithm operates directly on z-codes
- no need to look back at original values of LL and UR

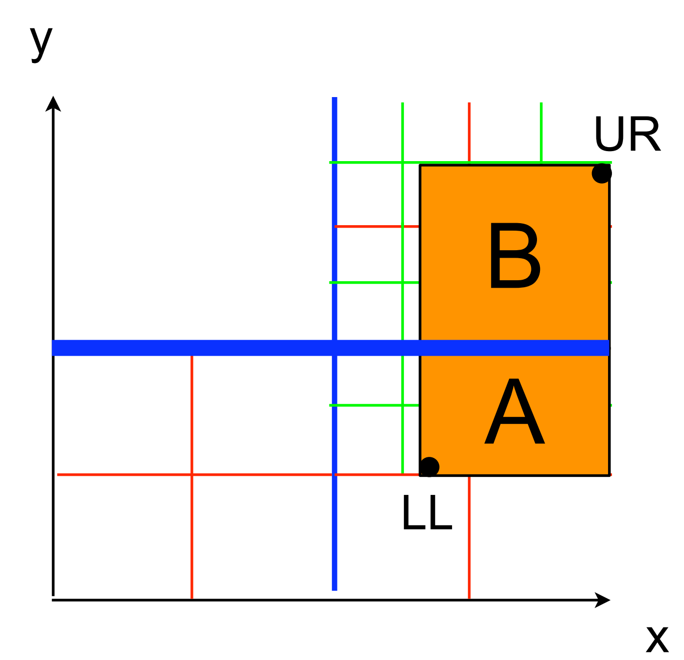
Direct Partitioning Example (1/3)

- example: LL
  x=5=101$_b$, y=2= 010$_b$
  z=38=100110$_b$
- UR
  x=7=111$_b$, y=6= 110$_b$
  z=62=111110$_b$
- bits equal
- => we did not hit the split line
- => no split happens!

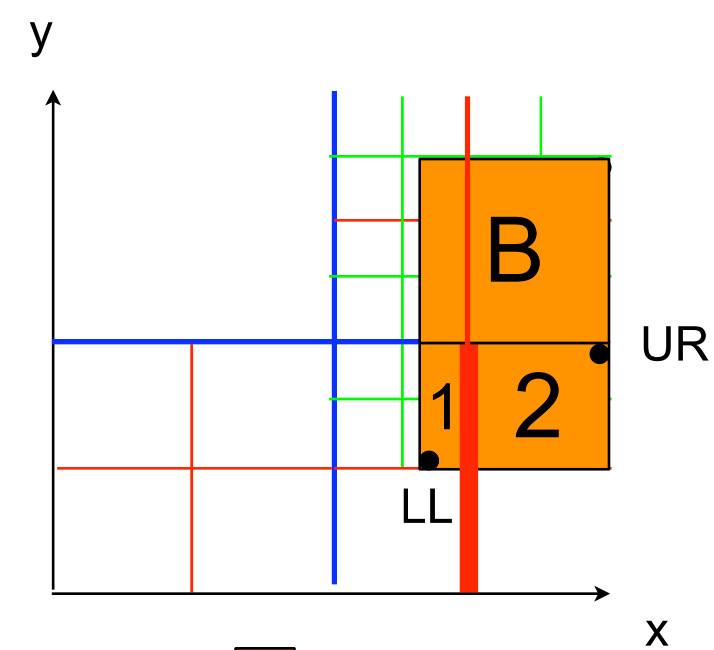


Direct Partitioning Example (2/3)

- example: LL
  x=5= 101_b, y=2= 010_b
  z=38= 100110_b
- UR
  x=7= 111_b, y=6= 110_b
  z=62= 111110_b
- bits differ
- => we have to split the z-interval
- [100110;111110] => [100110;101111], [110010;111110]
- [38,62] => [38;47], [50,62]
- thus we obtain the same result as above **in one step**



Direct Partitioning Example (3/3)



- Interval A:
- z_{low}= 38= 100110$_b$
  z_{high}=47= 101111 $_b$
- bits differ
- => we have to split the z-interval
- [100110;101111] => [100110;100111], [101100;101111]
- [38,47] => [38;39], [44,47]
- again: cut out something “in the middle“

Direct Splitting Rule

- we check dimensions in round-robin following the kd-trie partitioning scheme
- whenever we compare two numbers
  $z_{low} = \langle b_{low\ 1},\ldots,b_{low\ t}\rangle$ and $z_{high} = \langle b_{high\ 1},\ldots,b_{high\ t}\rangle$
  at position $1 \leq i \leq t$
- and $z_{low\ i} == z_{high\ i}$
- => we continue, i.e., move to the next split line
- otherwise, i.e. $b_{low\ i} \neq b_{high\ i}$, we split and compute the split z-interval:
- $[z_{low}\ ;\ \mathbf{z'_{high}}]$, $[\mathbf{z'_{low}};\ z_{high}]$
- $\mathbf{z'_{high}}$ = like z_{high}, however bit $b_{high\ i}$=0, $b_{high\ j}$=1 for j>i and j=i+2k, k>0, pattern ...0.1.1.1.1...
- $\mathbf{z'_{low}}$=like z_{low}, however bit $b_{low\ i}$=1,$b_{low\ j}$=0 for j>i and j=i+2k, k>0, pattern ...1.0.0.0.0....

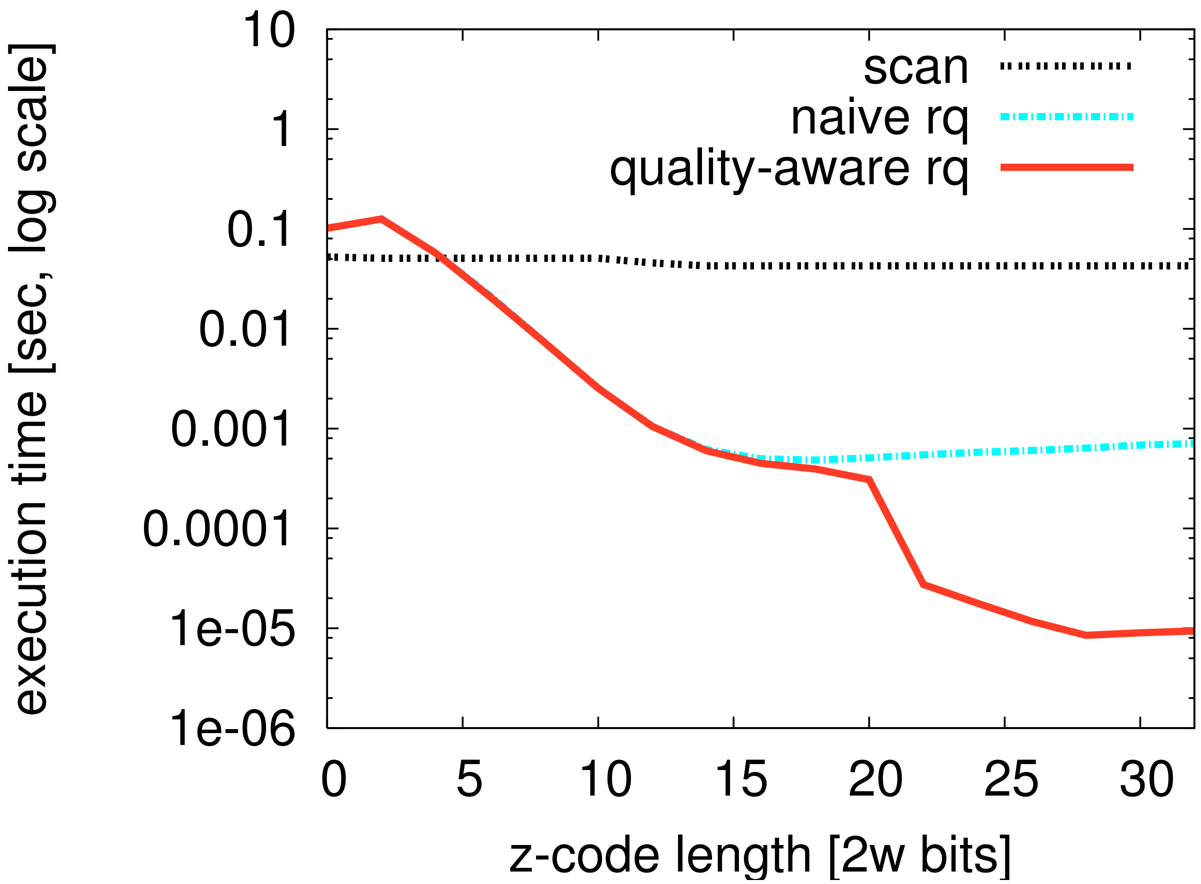
Discussion: Linearized B^+-Tree

- preserves all the good properties of the B+-tree (see previous chapters)
- only makes sense if efficient support of range queries is required
- very easy to integrate into existing systems
- easy extension of the insert-method
- easy extension of the find_key-method
- Note:
    - find_key-methods may return a superset
    - -> postfilter!
- already proposed in 1981, see Literature

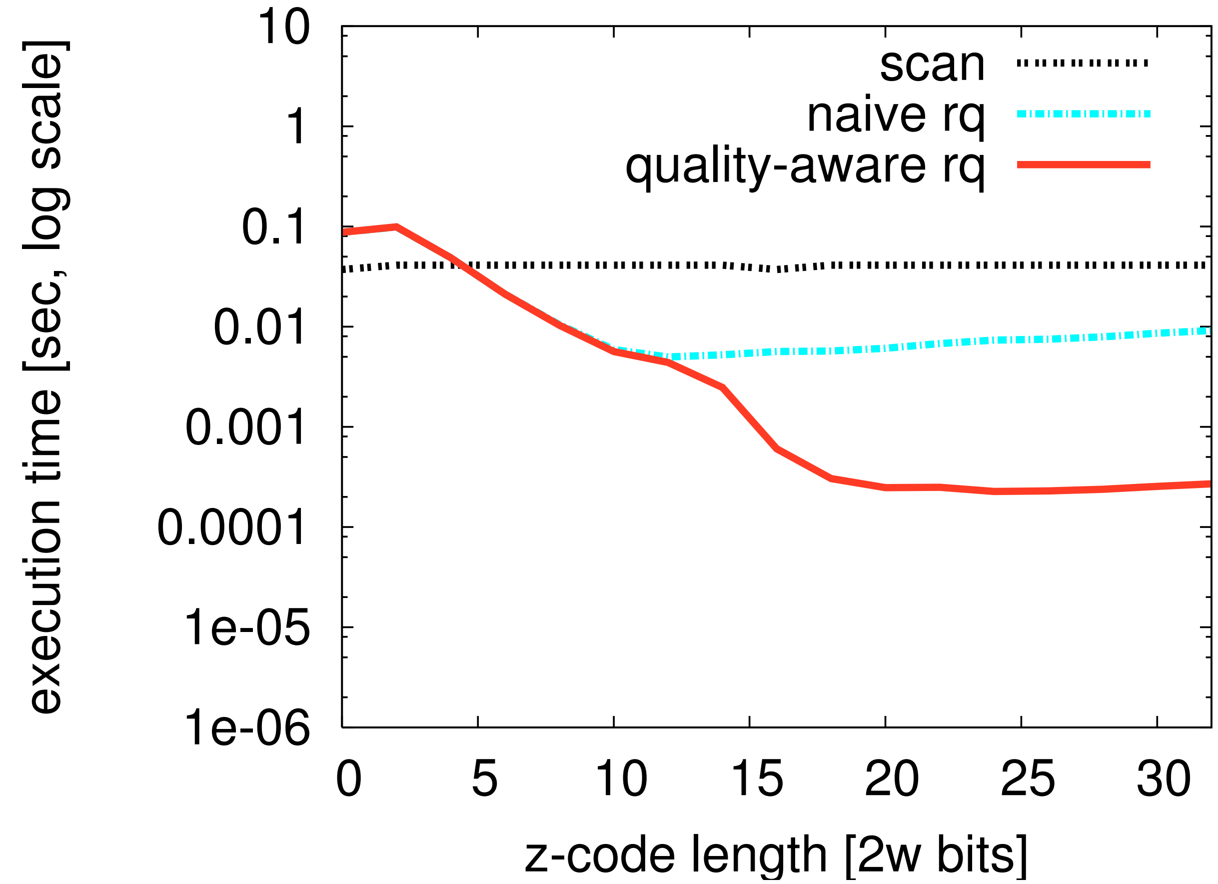
Wrap-up: Range Queries

- 1. approach: naïve
  - one interval query
  - considerable clipping: needs to postfilter many false positives
  - inefficient
- 2. approach: query partitioning
  - split query into subqueries
  - less clipping: needs to postfilter less elements
  - efficient if number of subqueries is not too high
- Note: for both approaches we have to postfilter results!

Experimental Results



window size = 1000



window size = 10,000

- index size: 10 million entries
- one computing core
- source: Jens Dittrich

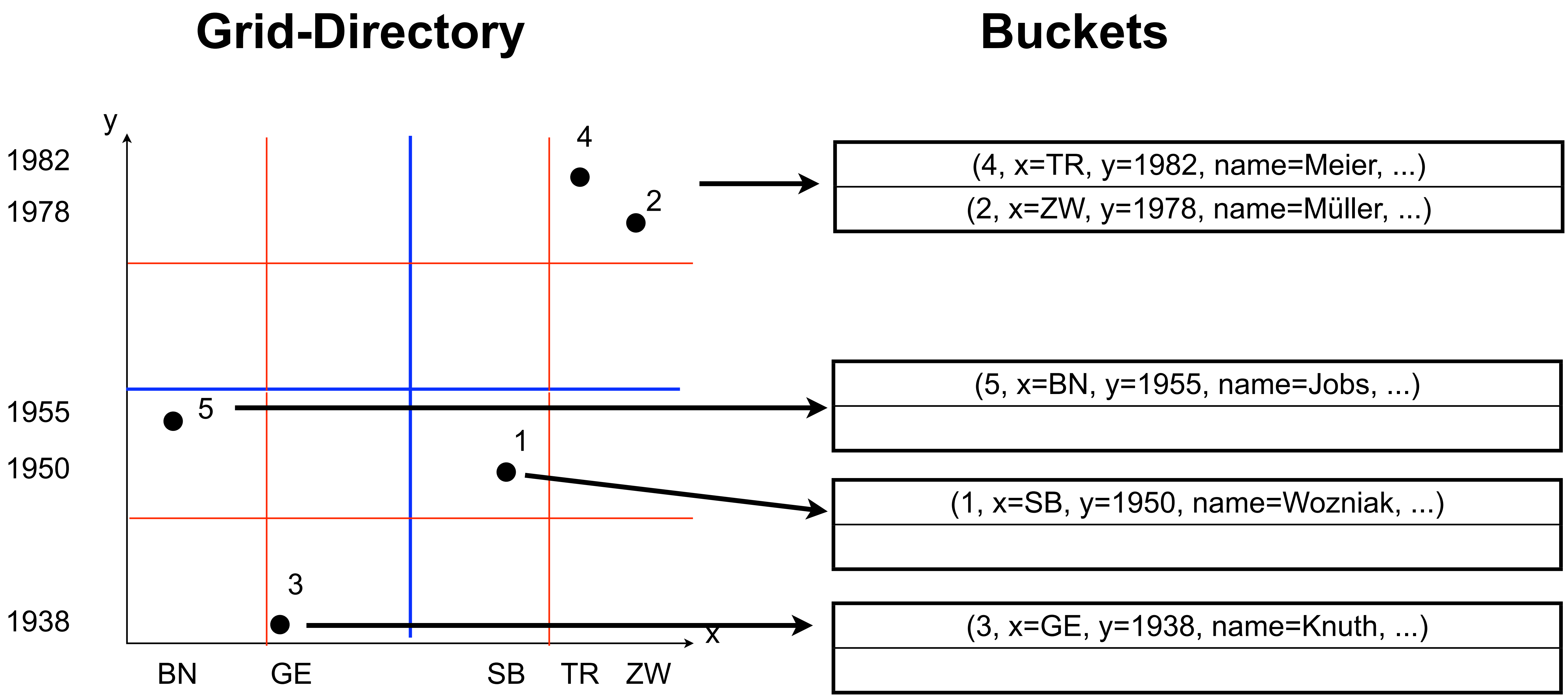
Multi-dimensional Indexes.

Grid-Indexes.

Grid-Indexes

- Idea: provide a **physical** partitioning of the data of a given granularity
- partition data space using a grid
- each grid cell points to a page on disk
- very similar to extendible hashing
- works very well for point data

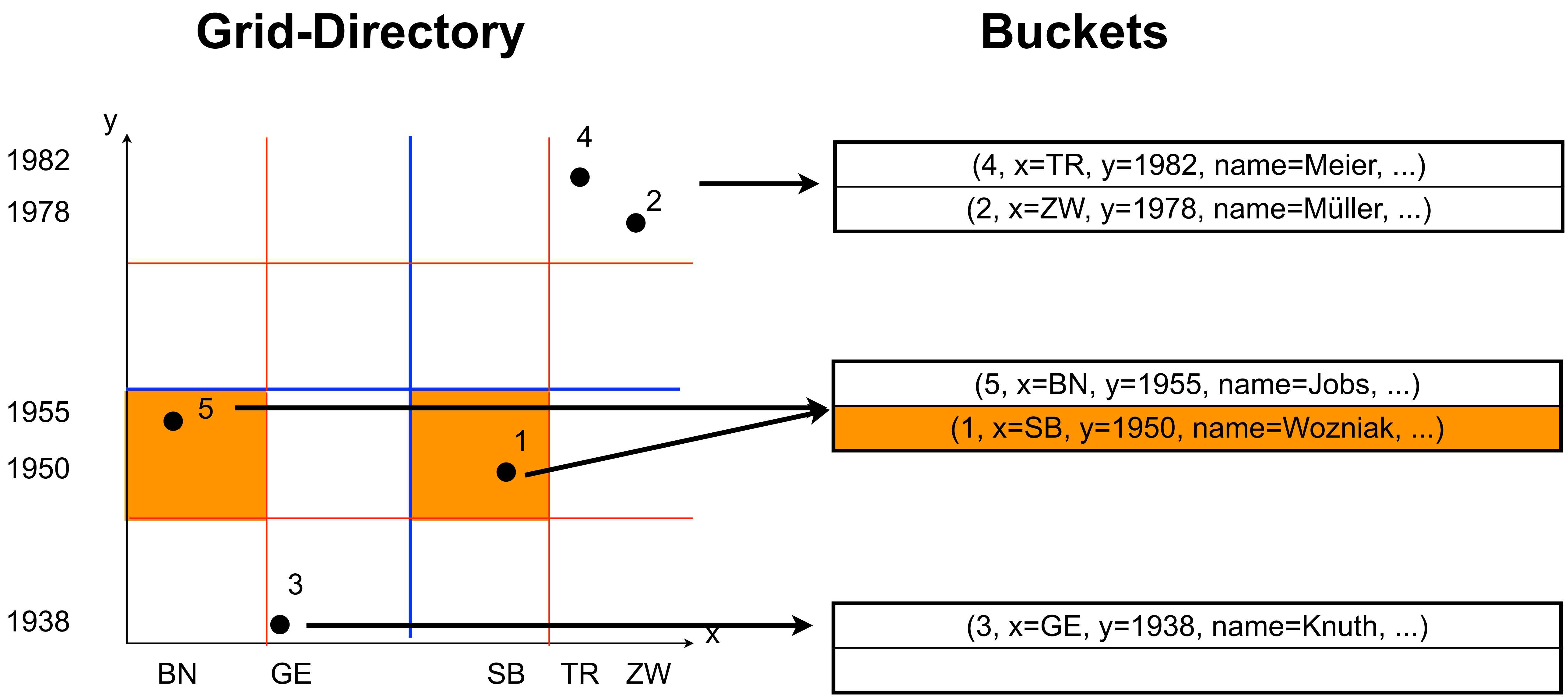
Grid-Index Example



- linear scale on both axes
- simplifies grid-cell computation
- grid-directory may reside on external storage

1: (Wozniak, 1950, SB)
2: (Müller, 1978, ZW)
3: (Knuth, 1938, GE)
4: (Meier, 1982, TR)
5: (Jobs, 1955, BN)

Grid-Index Example 2



- should merge buckets to achieve better space utilization
- however this may make matters complicated

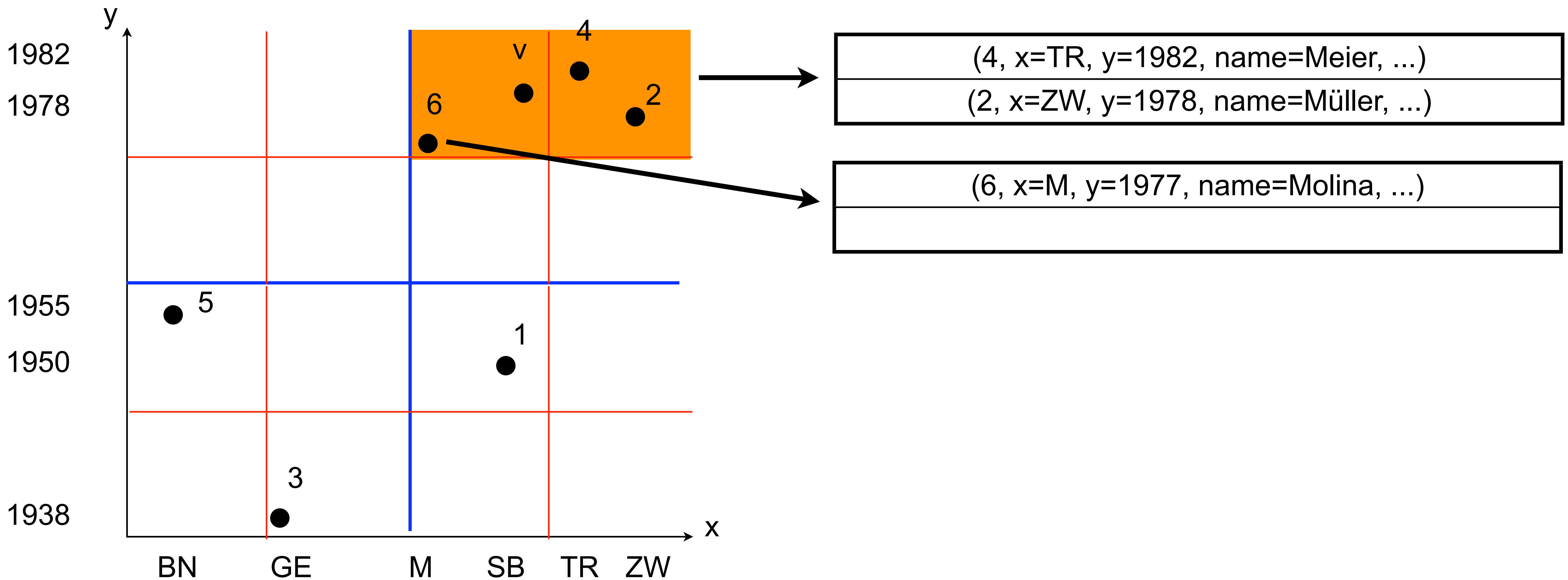
Properties

- Grid-directory may reside on disk
- linear scales are kept in main memory
- in most cases only 2 I/Os to answer a point query
    - one to fetch directory entry
    - one to fetch data page
- column or row-wise layout for directory
- => reduces I/O-effort to fetch directory entry

Range-Queries

- if query region intersects multiple cells
  => retrieve all these cells
  => depending on data layout may trigger considerable random I/O
- same problem for inserts: how to insert an object intersecting multiple regions?
- solution: insert the same object multiple times, query time: ??
- may lead to considerable redundancy
- grid-size should be adjusted to typical workload
- grid-cells too big => too much data to be searched at query time
- grid-cells too small => too much redundancy for inserts as well as to many cells to be retrieved for querying

NN-Queries



- given query point v
- retrieve bucket for v
- points found in that bucket may be **candidates**
- NN(v) would return 6, however 4 is the correct result
- therefore: adjacent cells have to be considered

Splitting Grid-Cells

- Grid-cells may be split similar to cells in a tree or a directory in extendible hashing
- Idea: start with an initial grid partitioning
- if buckets overflow: split and re-assign elements
- delete also possible => merge buckets

Grids Versus Ext. Hashing Versus z-Codes

- Grids, extendible hashing, and z-codes are in fact very similar
- Grid:
  - interprets hash as multi-dimensional numbering of the space
  - tries to preserve dimension-wise locality among neighboring cells **either** in row-wise **or** in column-wise order
- Extendible Hashing:
  - uses **last** k bits instead of **leading** k bits
  - goal: declustering -> opposite of keeping spatial locality
- z-codes:
  - interprets hash as multi-dimensional numbering of the space
  - tries to preserve spatial locality using a space-filling curve

z-Code Grid

- why not use z-code numbering to build a grid of a given granularity?
- same method as grid-file
- however: space filling curve for cell numbering
- each cell maps to a bucket/page
- split if necessary, i.e., add a bit to the existing prefix
- in the end this would be another variant of a tree-structured index....
- so why not use a linearized B^+-tree anyway?
- pros of Grid: works well for coarse-granular grid cells
- pros of B^+-tree: works well for both coarse granular and fine-granular grid cells

Wrap-Up

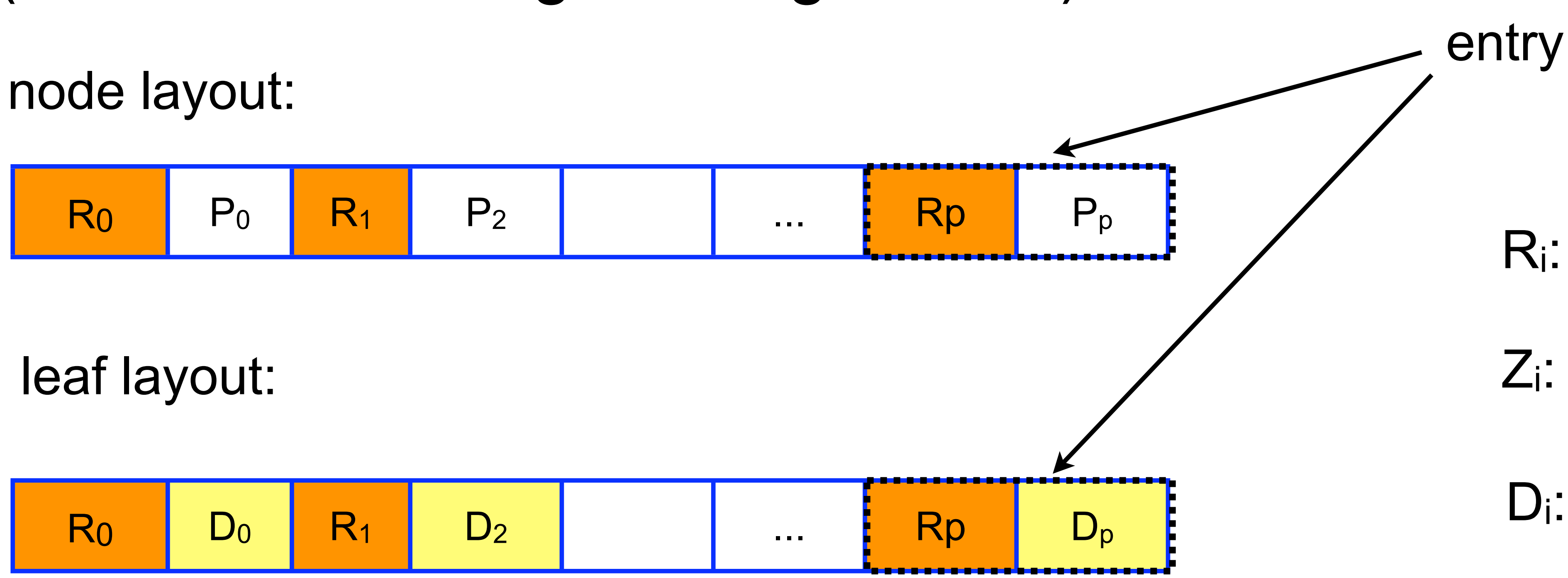
- learn to differentiate among **logical** and **physical** data partitioning
- all methods shown so far are very similar with respect to their logical partitioning
- however method differ w.r.t. their physical partitioning
- in practice: look at data and query/update workload in order to pick method
- optimal solution: method that is able to switch among different physical representations

Multi-dimensional Indexes.

R-trees.

R-Tree: Definition

- Very similar to B^+-trees
- each node has m entries ($M/2 \leq m \leq M$)
- subtree does **not** correspond to an interval anymore
- **but**: subtree corresponds to a rectangle R_i (minimal bounding rectangle/MBR)

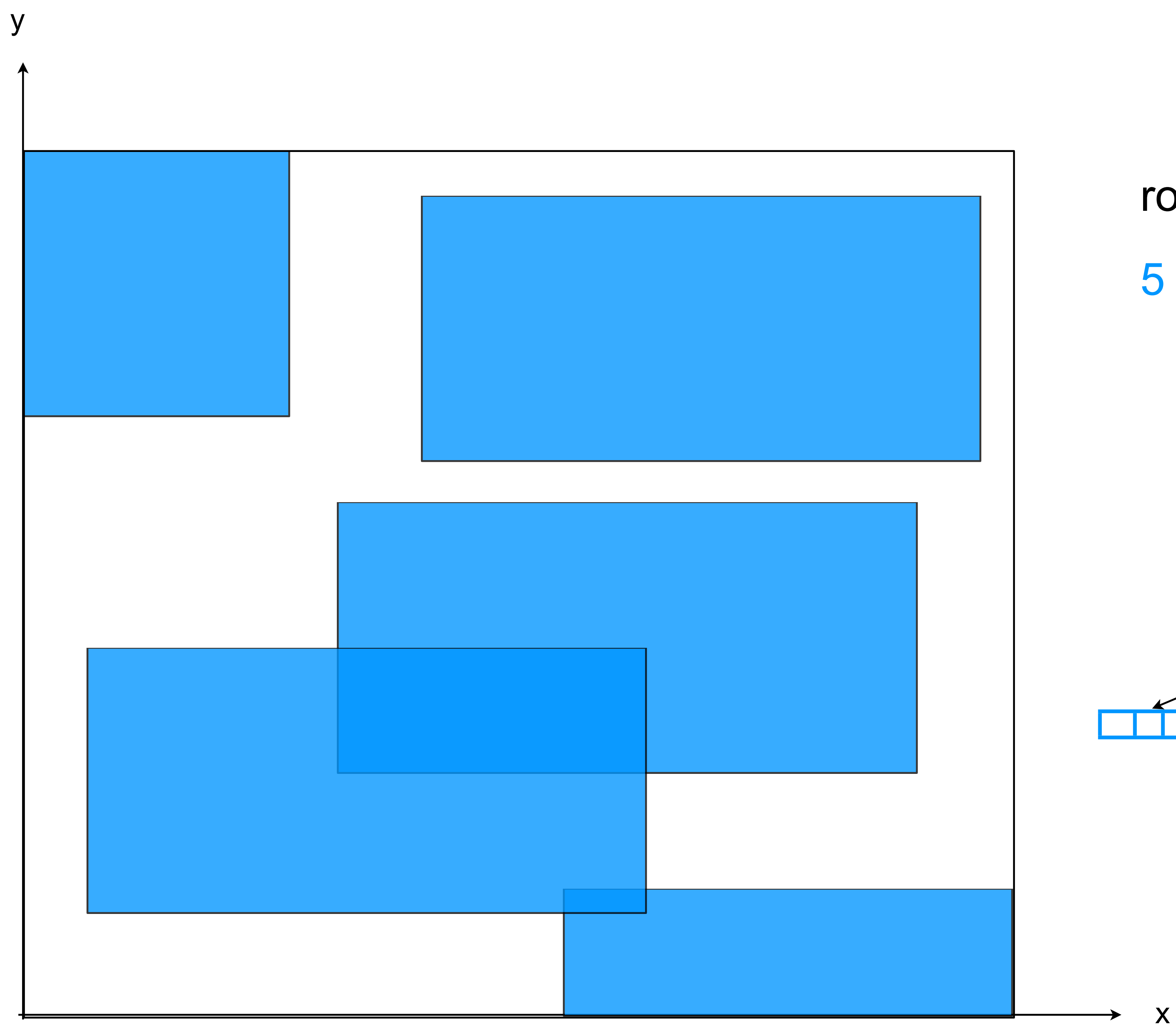


R_i: rectangle

Z_i: pointer

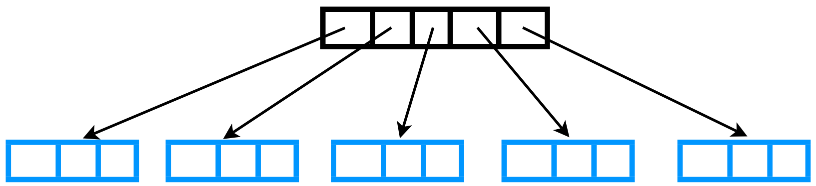
D_i: data entry

Example

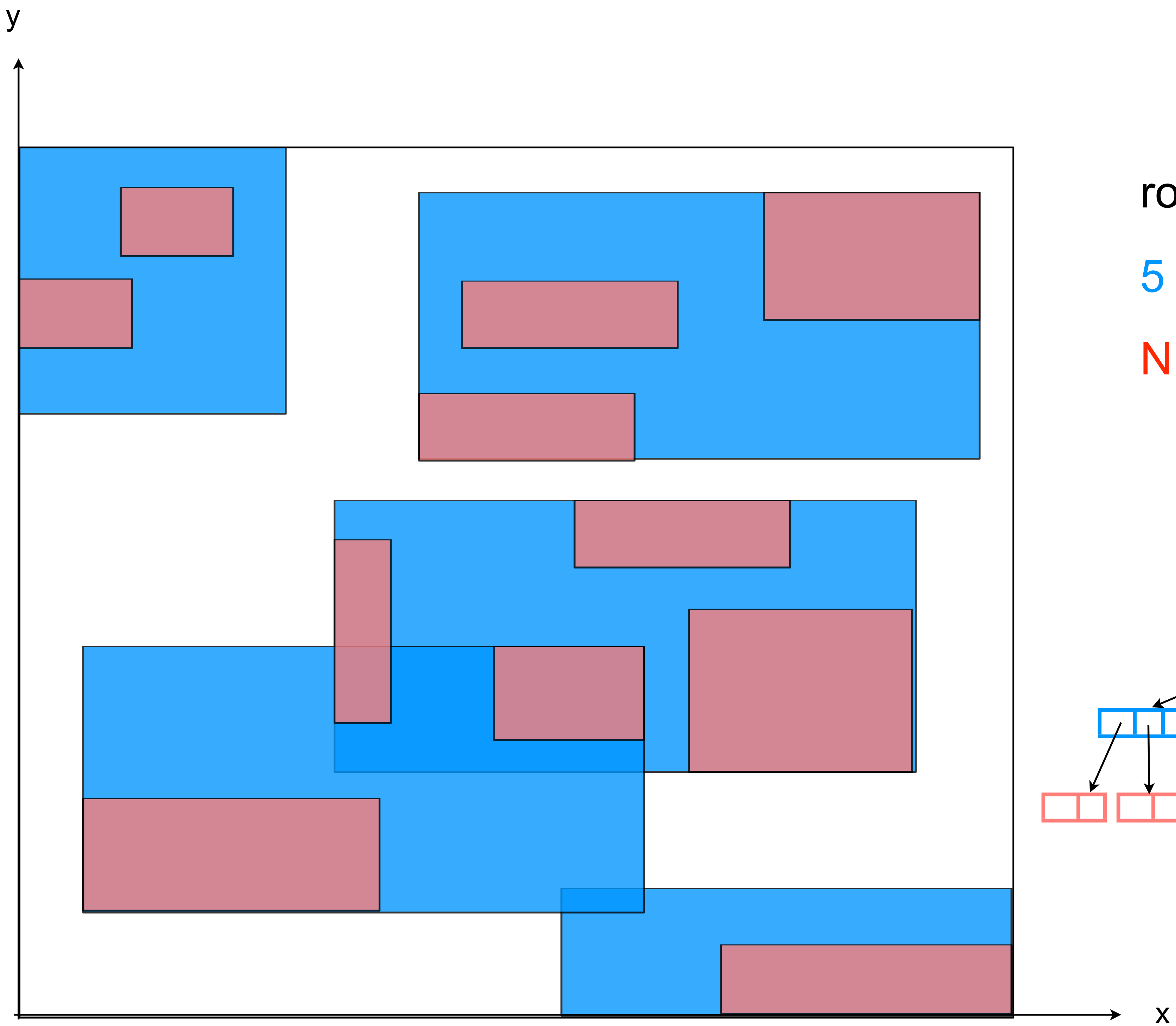


root

5 children



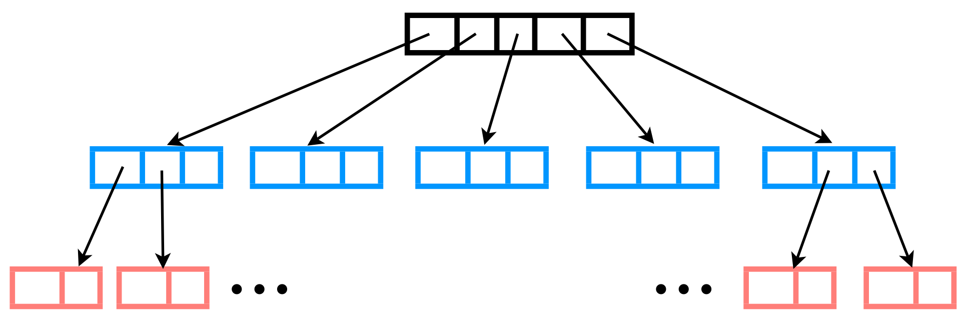
Example



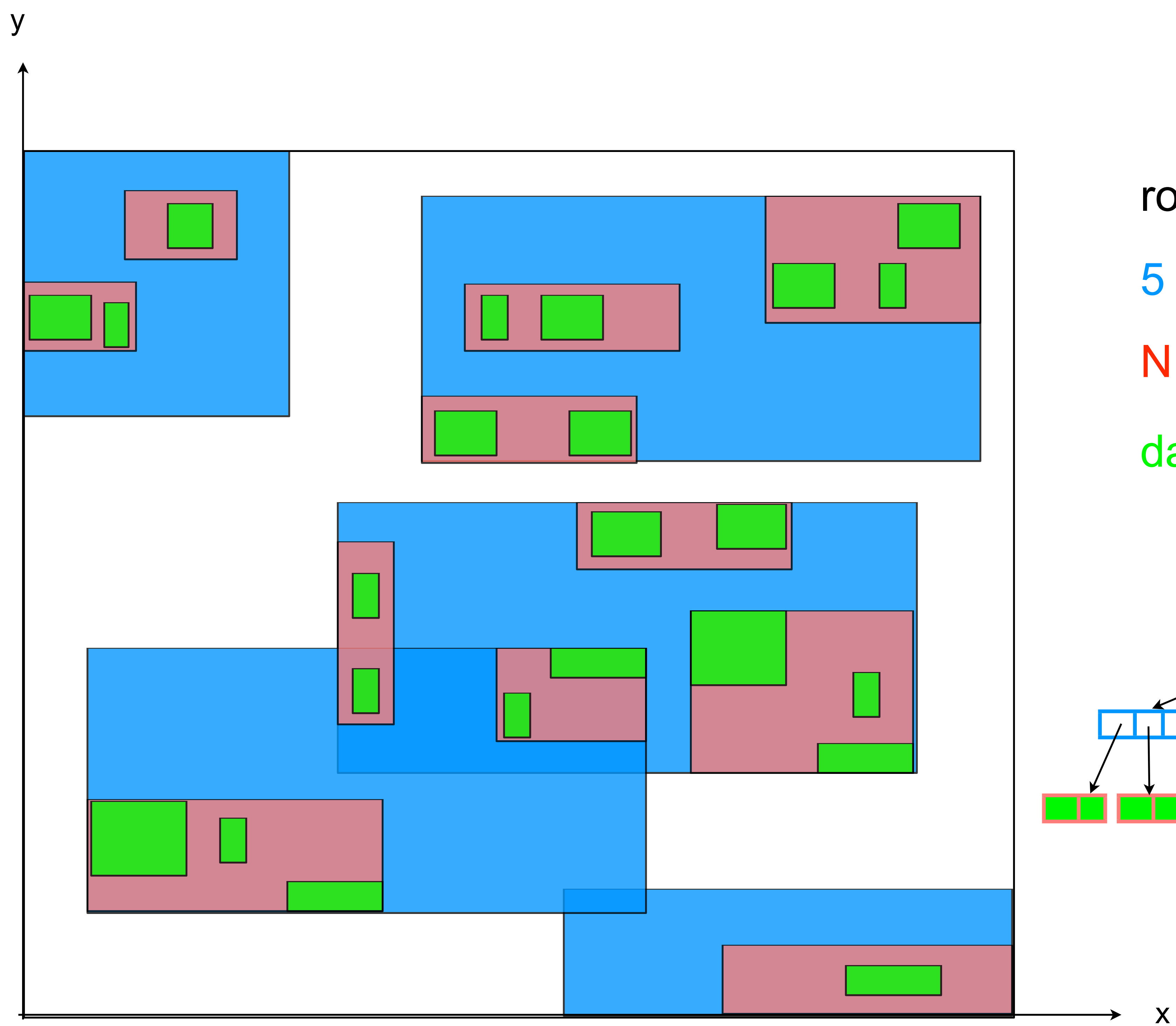
root

5 children

N grand-children



Example

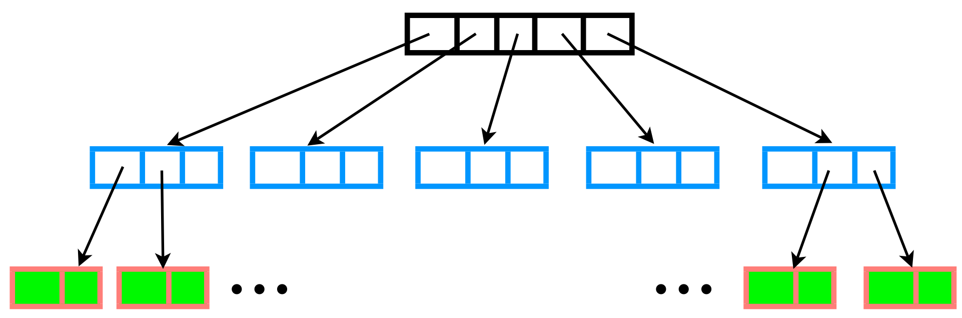


root

5 children

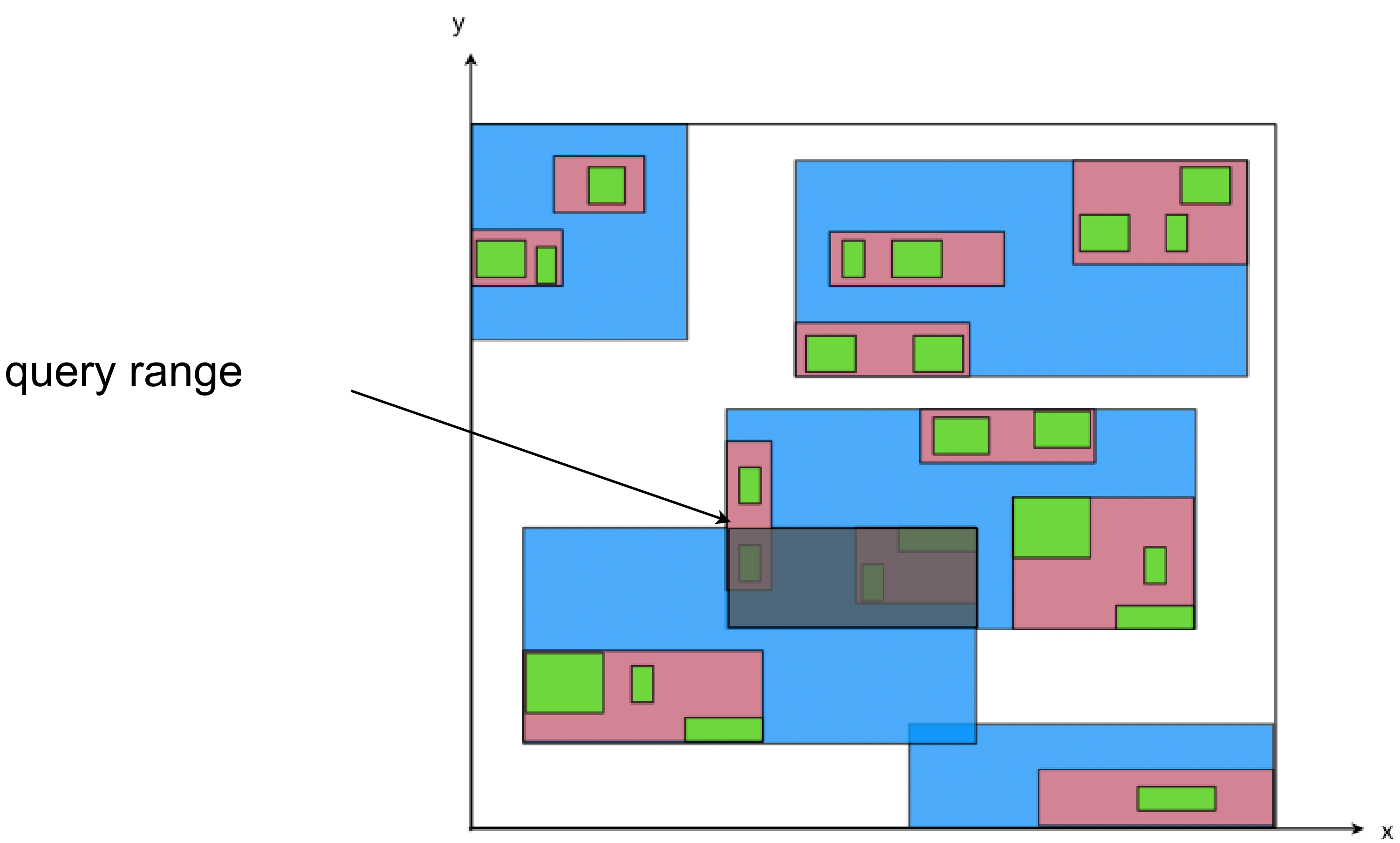
N grand-children

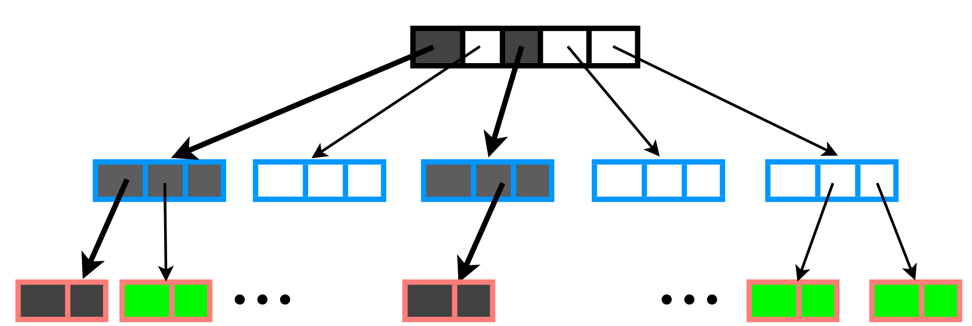
data



Properties of an R-Tree

- every path from the root to a leaf has the same length h
- rectangles inside a node/leaf do **not** have to be disjoint
  - during a find-operation it may occur that multiple subtrees have to be considered
  - strong overlap on all levels possible (more dimensions => more overlap)





R-Tree Splits

- Goal: decrease overlap of rectangles inside nodes
- R*-tree = R-tree optimizing split operations
- goal is reached by minimizing
  - **containment**: range that is covered by a node
  - **overlap**: range that is overlapped by two or more nodes
  - **circumference**: sum of the circumferences of all rectangles in this node
- the above parameters are optimized during a node split
- R*-tree is for many scenarios more efficient than the standard R-tree
- Literature:
  Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference 1990: 322-331.

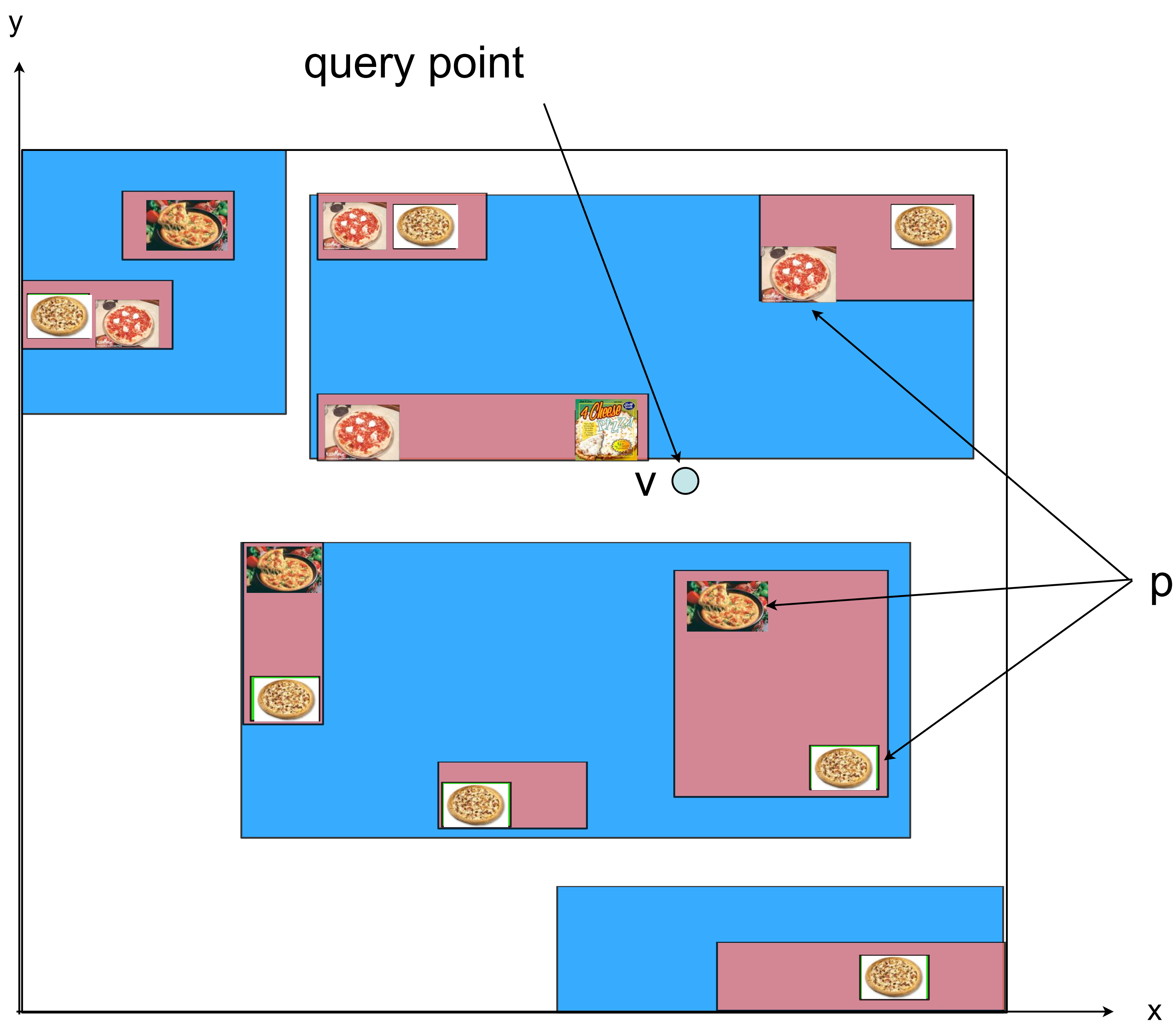
k-NN-Queries

- priority-driven algorithm
- instead of traversing the tree using preorder, level-wise, etc
  => use priority
- recursion breaking:
  - stack -> preorder
  - queue -> level-wise
  - heap -> this algo
- works for many index structures

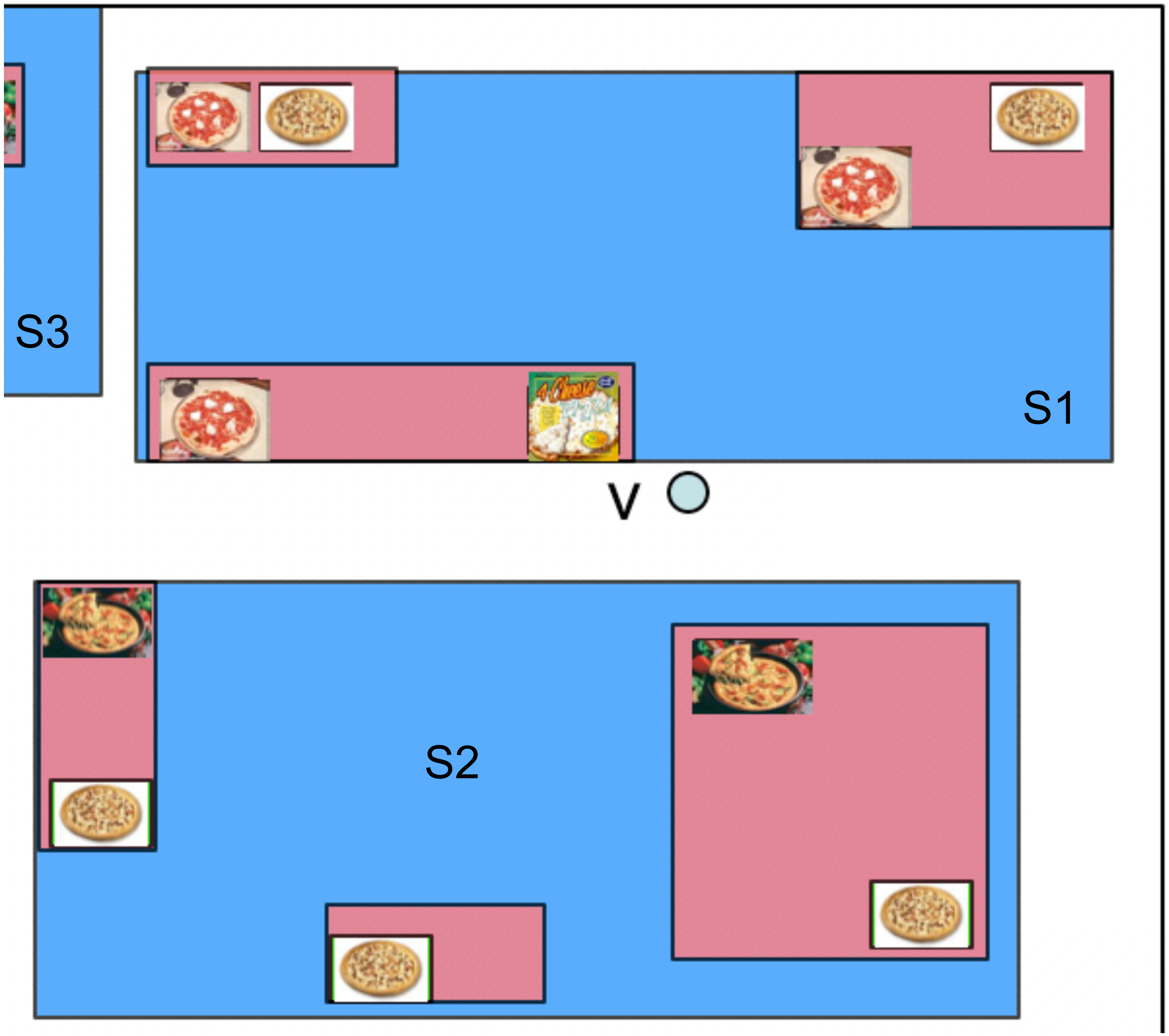
```
k-NN-Algorithm (QueryPoint v, NumberOfResults k)
result_count = 0
pq = priority_queue
pq.insert(root, 0)

While (result_count < k):
    top = pq.pop()
    If (top is a node):                       Unfold node
        ForEach child of top:
            d = distance(child, v)
            pq.insert(child, d)
        EndForEach
    ElseIf (top is a leaf):                   Unfold leaf
        ForEach data_entry of top:
            d = distance(data_entry, v)
            pq.insert(data_entry, d)
        EndForEach
    Else // i.e., top is a data entry:        Report result
        report top as a result
        result_count ++;
EndWhile
```

k-NN-Queries (Pizza-Queries)



k-NN-Queries (Pizza-Queries)



Unfold root

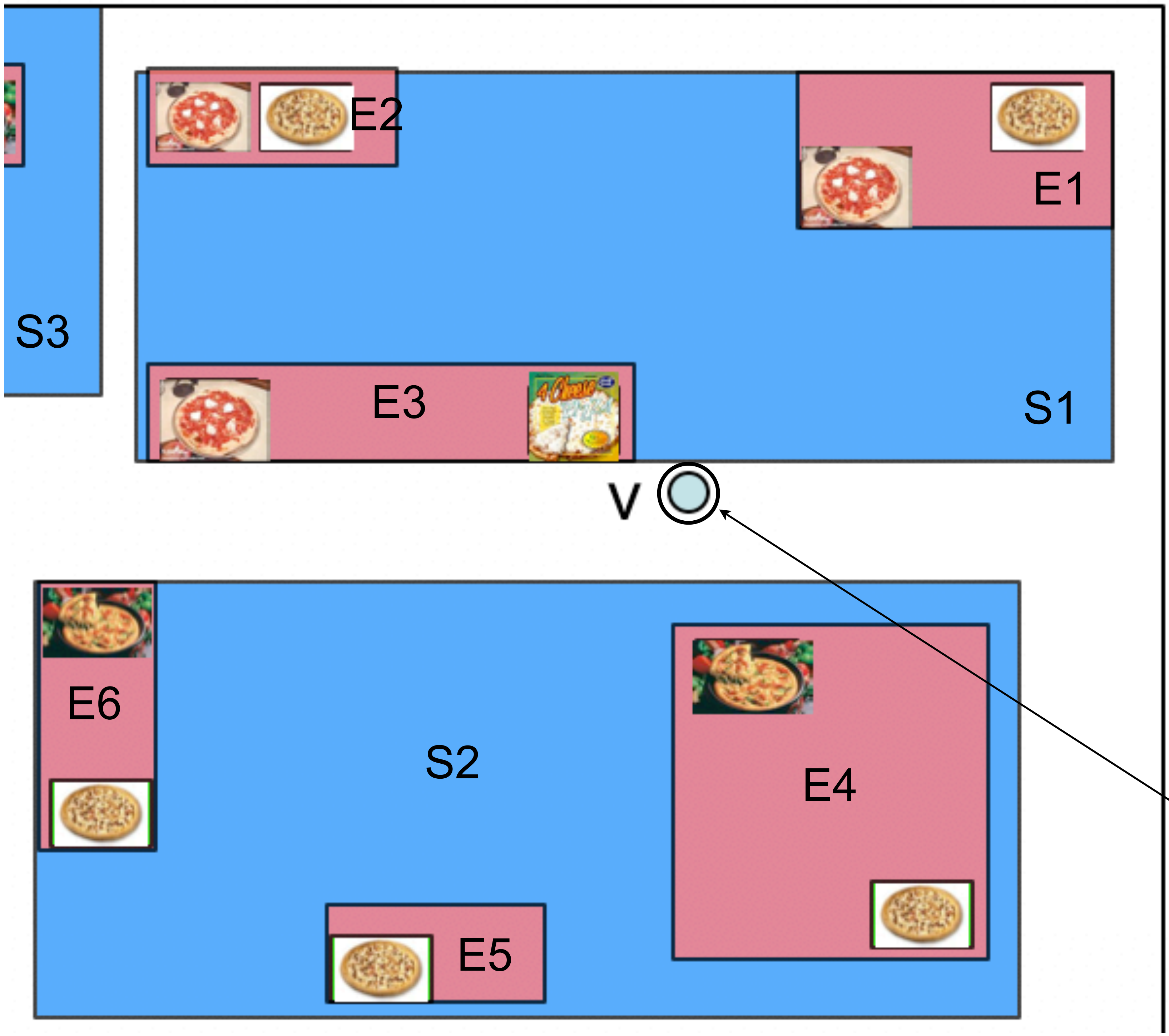
priority queue pq:

d(v,S1), S1

d(v,S2), S2

d(v,S3), S3

k-NN-Queries (Pizza-Queries)

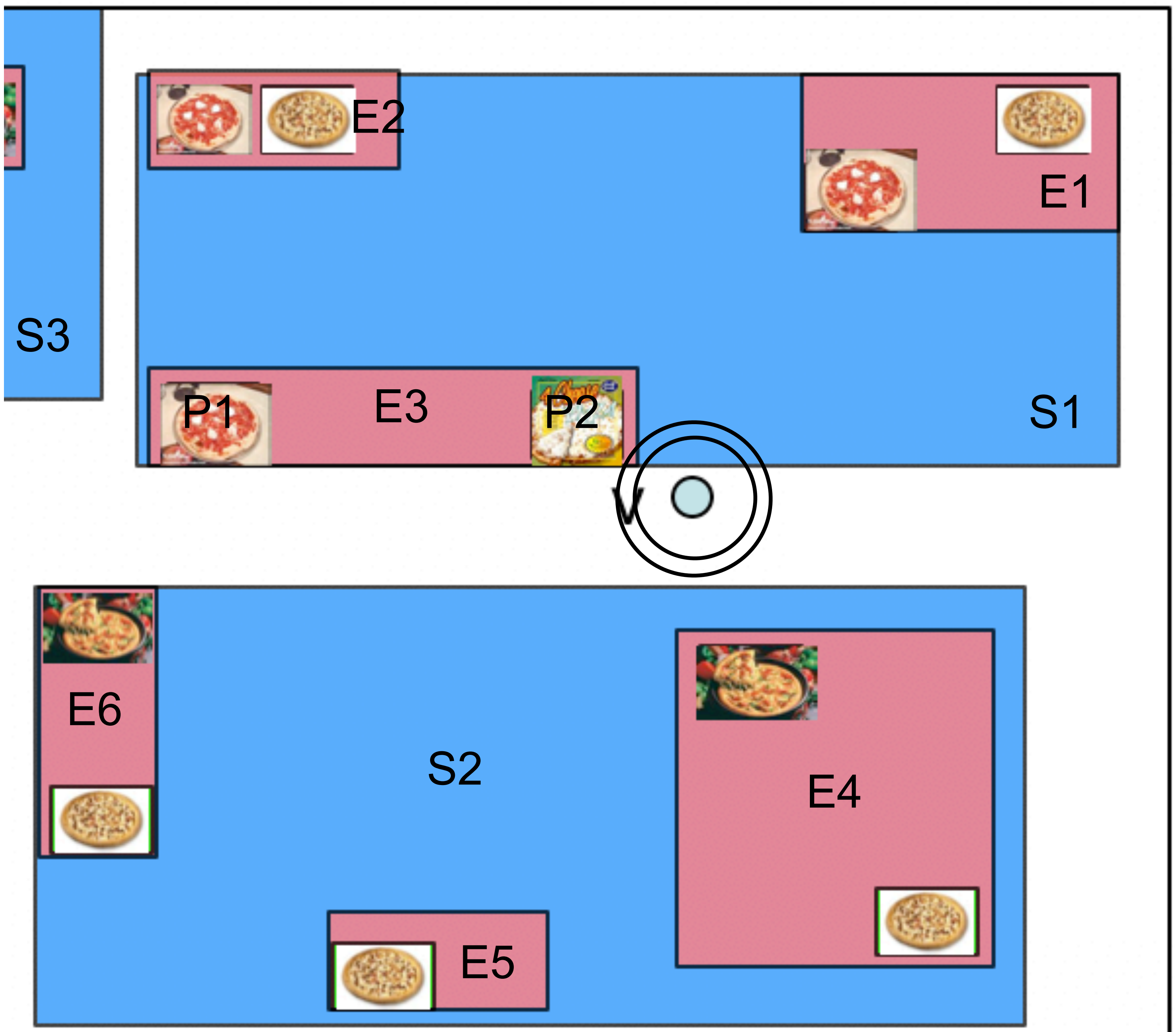


Unfold root
priority queue pq:
d(v,S1), S1
d(v,S2), S2
d(v,S3), S3

Unfold S1
priority queue pq:
d(v,E3), E3
d(v,S2), S2
d(v,E1), E1
d(v,E2), E2
d(v,S3), S3

current distance of the top-element in pq to v

k-NN-Queries (Pizza-Queries)

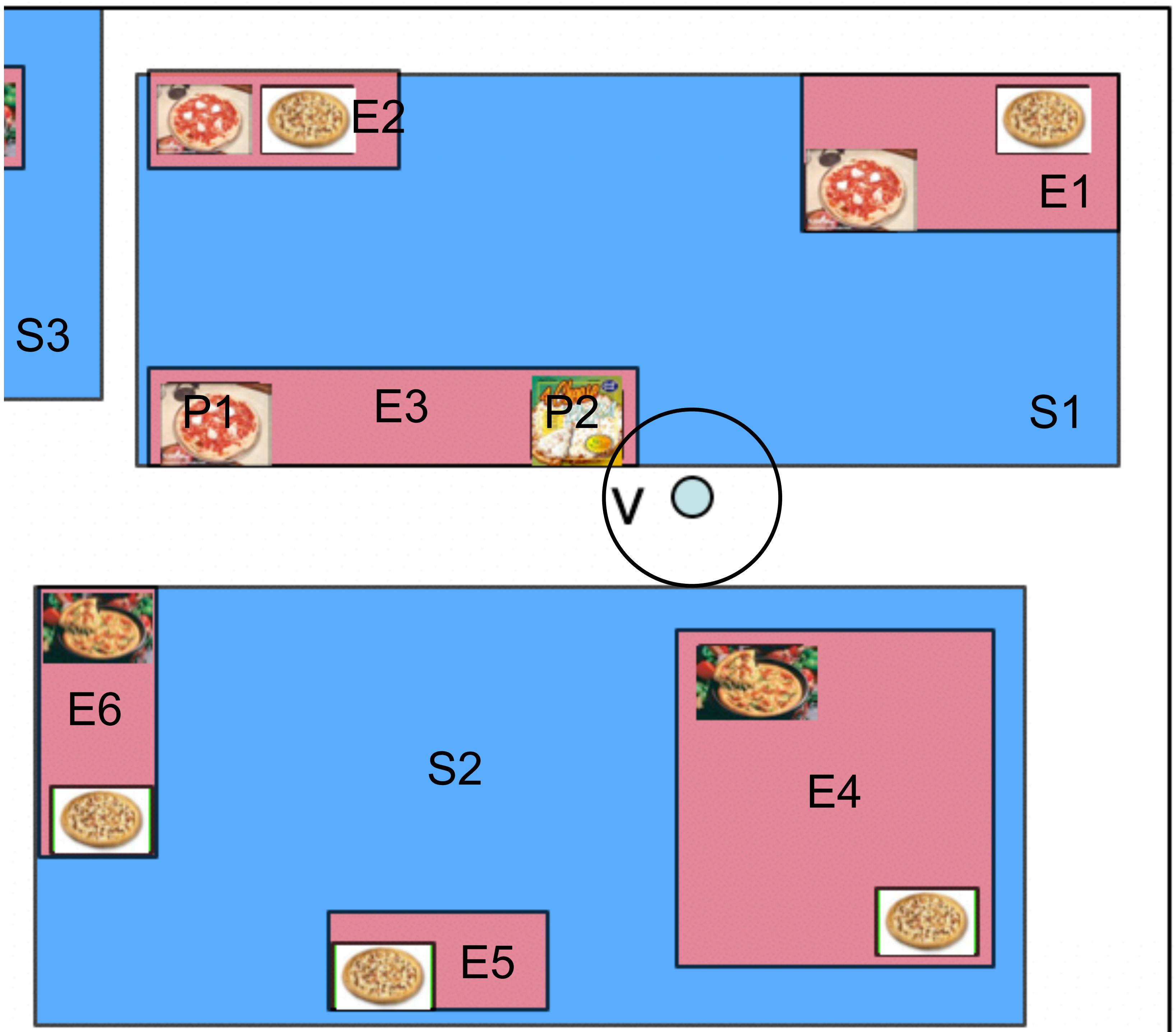


Unfold E3
priority queue pq:
d(v,P2), P2
d(v,S2), S2
d(v,E1), E1
d(v,P1), P1
d(v,E2), E2
d(v,S3), S3

Result P2
priority queue pq:
d(v,S2), S2
d(v,E1), E1
d(v,P1), P1
d(v,E2), E2
d(v,S3), S3

1-NN: P2

k-NN-Queries (Pizza-Queries)

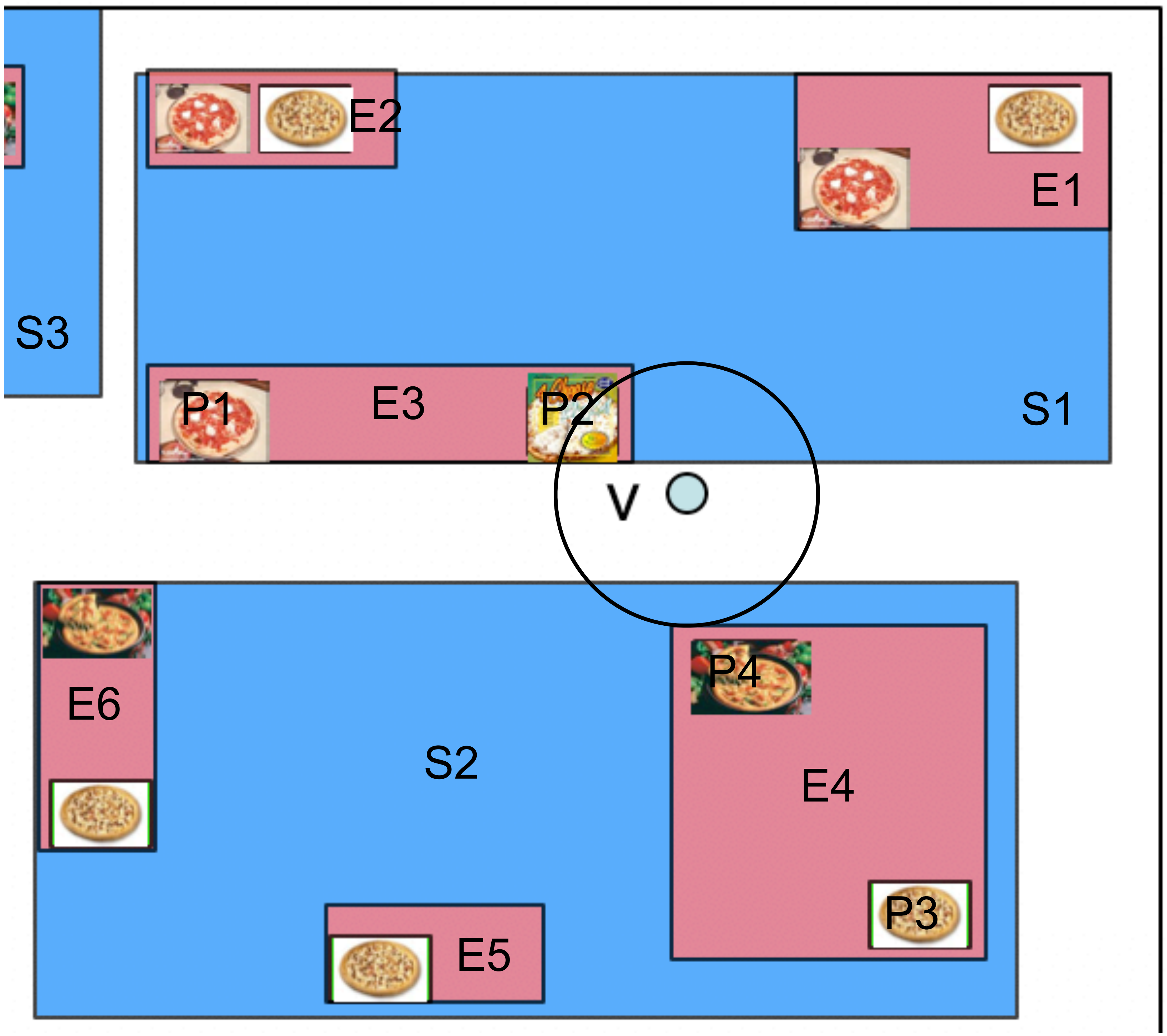


Unfold S2

priority queue pq:

d(v,E4), E4
d(v,E1), E1
d(v,P1), P1
d(v,E5), E5
d(v,E2), E2
d(v,E6), E6
d(v,S3), S3

k-NN-Queries (Pizza-Queries)

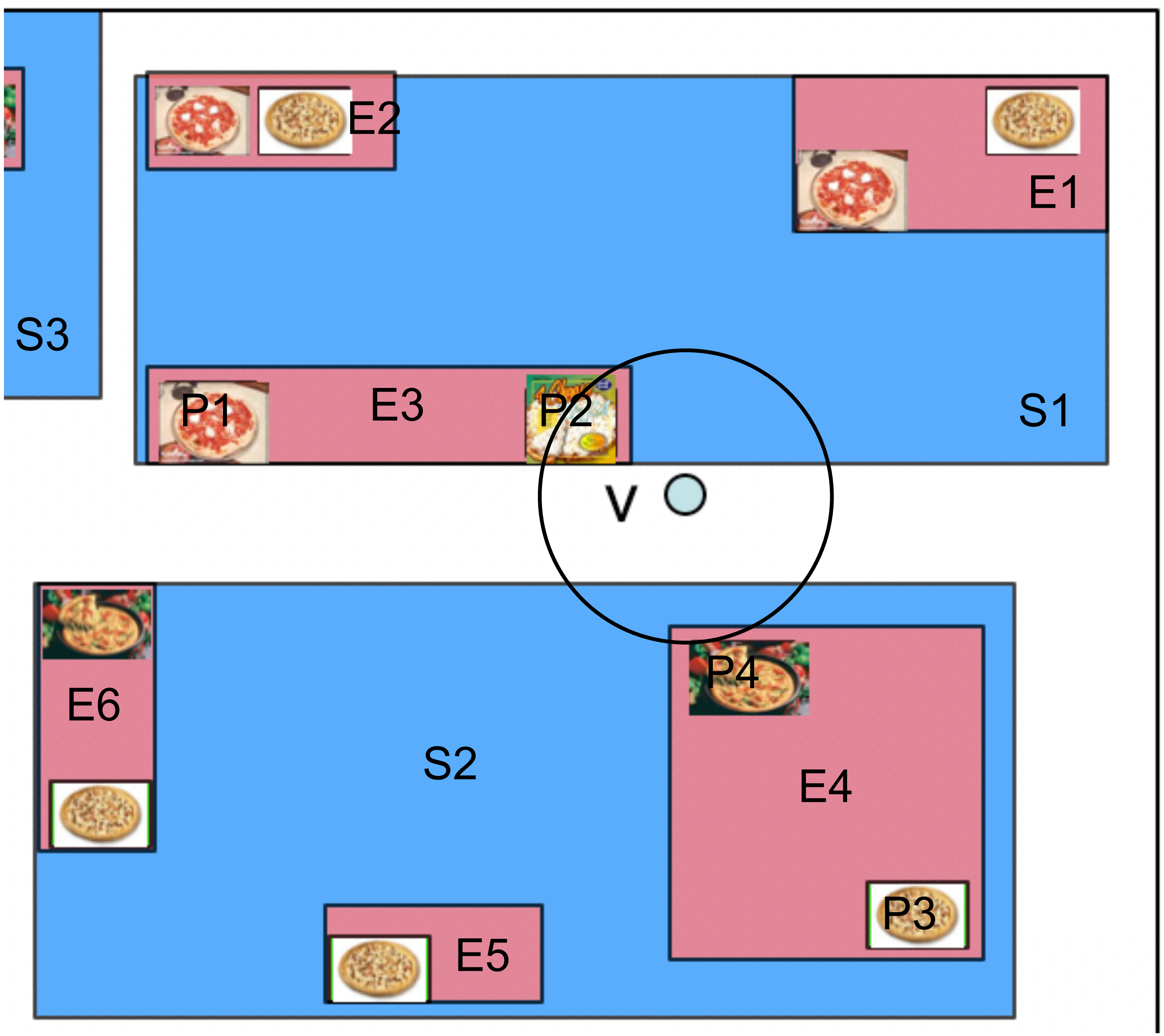


Unfold E4

priority queue pq:

d(v,P4), P4
d(v,E1), E1
d(v,P1), P1
d(v,P3), P3
d(v,E5), E5
d(v,E2), E2
d(v,E6), E6
d(v,S3), S3

k-NN-Queries (Pizza-Queries)



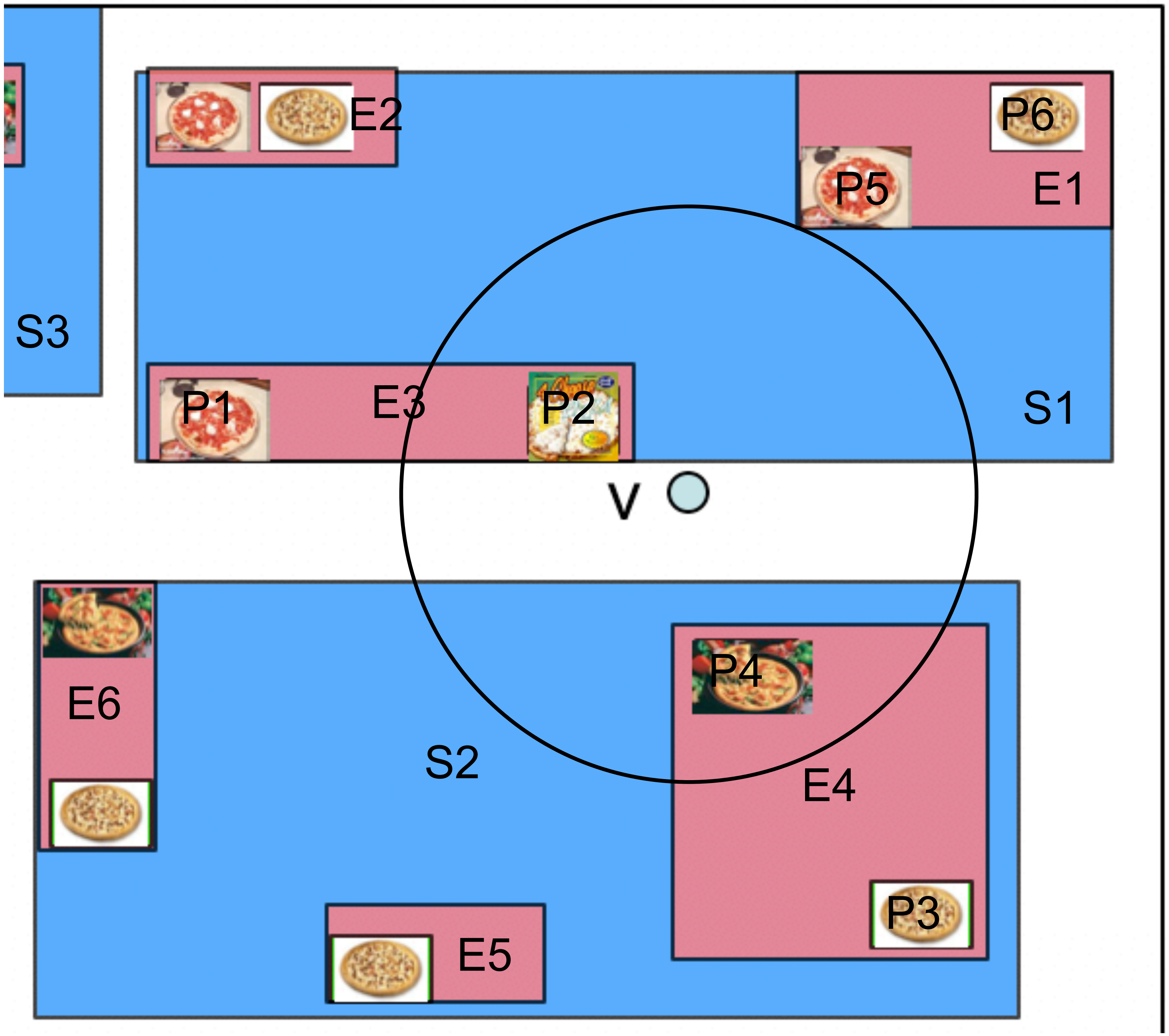
Result P4

priority queue pq:

d(v,E1), E1
d(v,P1), P1
d(v,P3), P3
d(v,E5), E5
d(v,E2), E2
d(v,E6), E6
d(v,S3), S3

2-NN: P4

k-NN-Queries (Pizza-Queries)

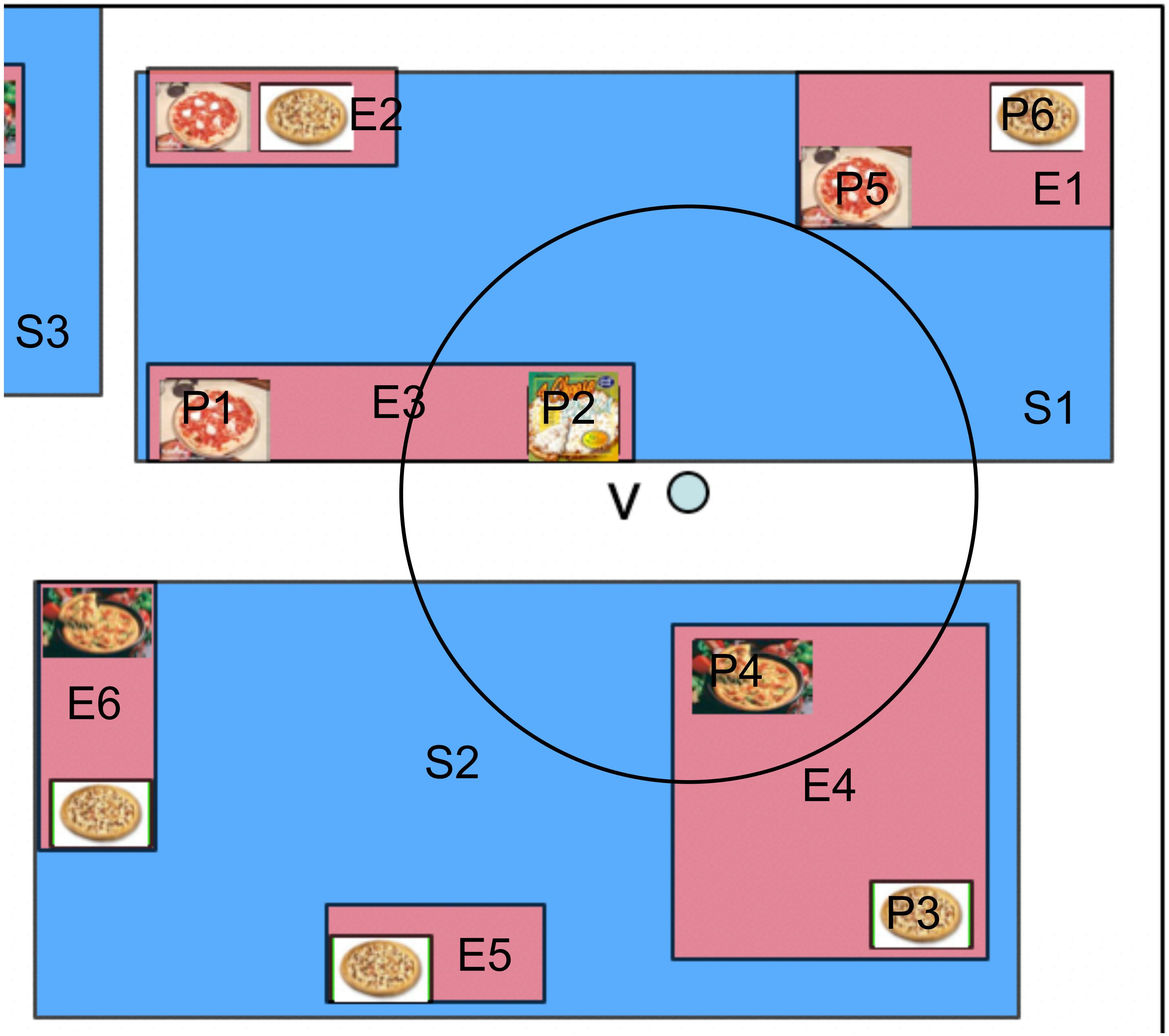


Unfold E1

priority queue pq:

d(v,P5), P5
d(v,P1), P1
d(v,P3), P3
d(v,E5), E5
d(v,E2), E2
d(v,P6), P6
d(v,E6), E6
d(v,S3), S3

k-NN-Queries (Pizza-Queries)



Result P5

priority queue pq:

d(v,P1), P1
d(v,P3), P3
d(v,E5), E5
d(v,E2), E2
d(v,P6), P6
d(v,E6), E6
d(v,S3), S3

3-NN: P5

Discussion

- node processing is based on priority (distance of nodes to query point)
  - only relatively few nodes have to be visited
- k results are produced in correct order
- after k results traversal stops
- this algorithm works for many index structures
- e.g. kd-trees/tries

Multi-dimensional Indexes.

Generalized Tree Indexes.

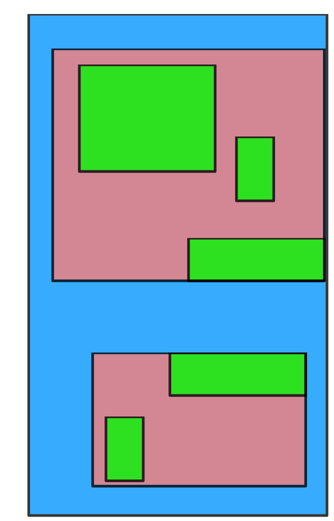
GIST

- Observation: R-tree is “a variant“ of the B^+-tree
- What exactly does “a variant” mean?
- difference to B^+-tree:
  - subtrees are characterized by minimal bounding rectangles (instead of disjoint intervals)
  - other split-strategy
  - during search traversal of multiple subtrees possible
  - number of node entries: m entries ($M/2 \leq m \leq M$)
- similarities with B^+-tree:
  - nodes/leaves correspond to pages
  - insert/delete/split similar

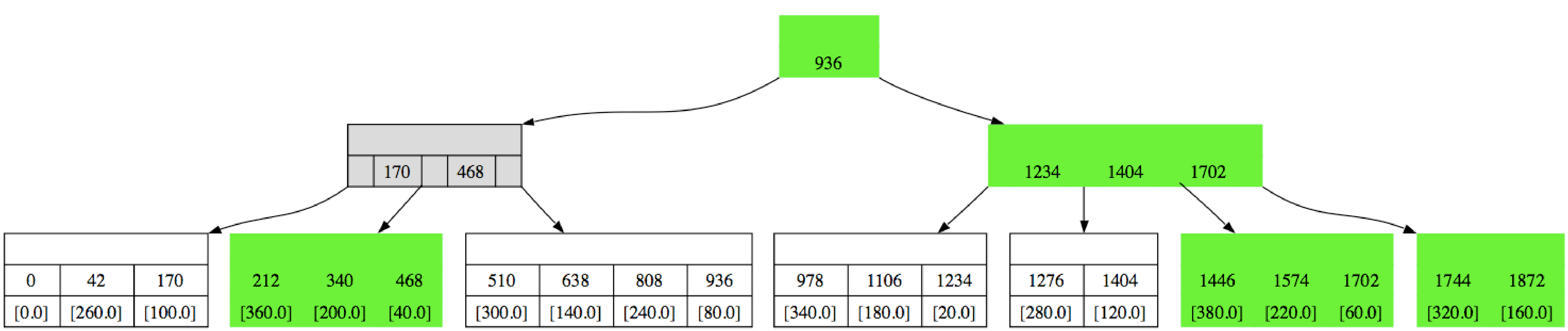
GIST: Predicate Tree

- 1. Idea:
  - characterize data of a subtree by a predicate
  - the predicate of a node is fullfilled by all nodes/leaves/data of the subtree

    ⇨ covering predicates

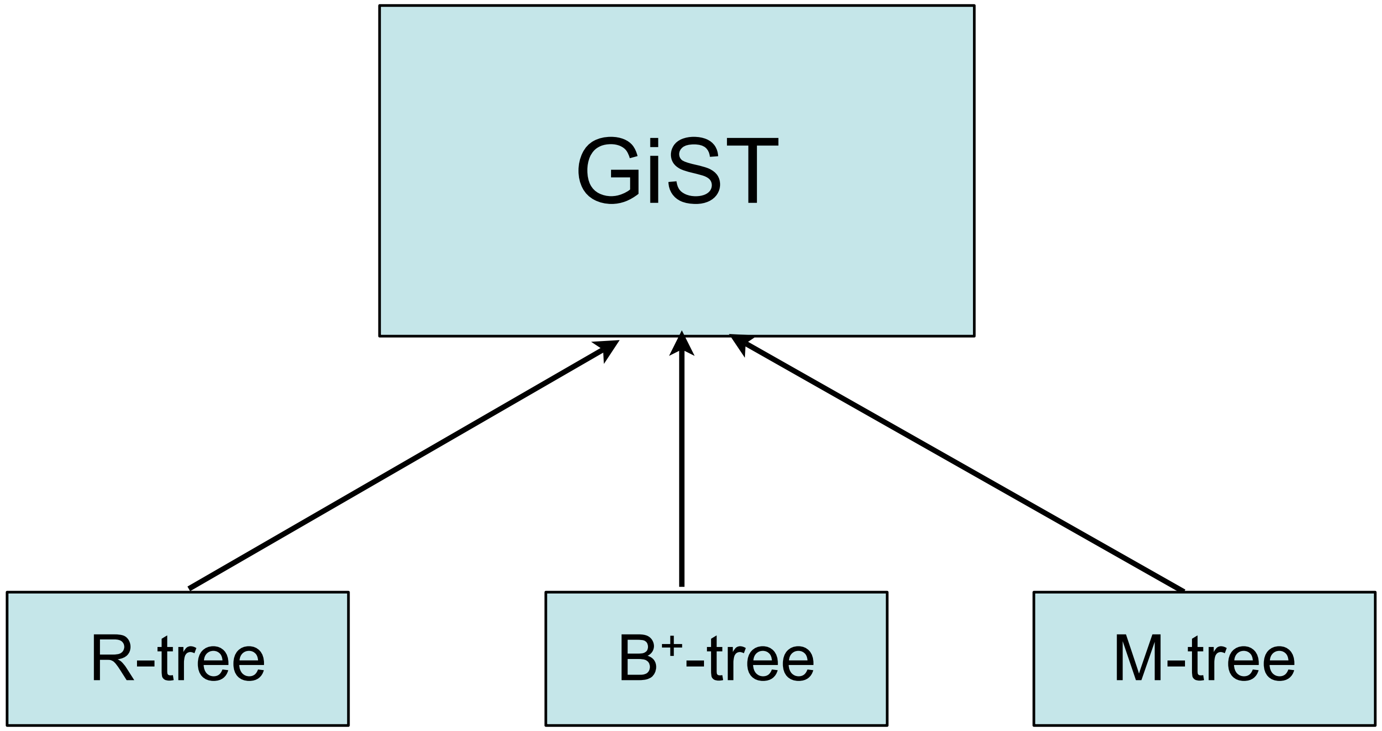


- Recall:
  - R-tree: predicate = data inside a rectangle R_i
  - B^+-tree: predicate = data inside an intervals $(S_i;S_{i+1}]$



GIST Framework

- 2. Idea:
  - provide a framework for the implementation of predicate-based index structures
  - special cases can then be implemented by providing or extending appropriate split/insert-strategies



GIST

- Literature:
  Joseph M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer: Generalized Search Trees for Database Systems. VLDB 1995: 562-573
- Other approach:
  Jochen Van den Bercken, Jens-Peter Dittrich, Bernhard Seeger: javax.XXL: A prototype for a Library of Query processing Algorithms. SIGMOD Conference 2000: 588

Literature on Index Structures

- Volker Gaede, Oliver Günther: Multidimensional Access Methods. ACM Computing Surveys 30(2): 170-231 (1998)
- hundreds of other index structures have been proposed since then
- in general
  - B+-tree plus extension will do the job
  - be skeptical whether new index helps
  - what benefit do they give over B+-trees?
  - how hard will they be to integrate into your system?
  - is the benefit they provide worth the implementation/integration effort?
  - does it make sense to use an index anyway?....

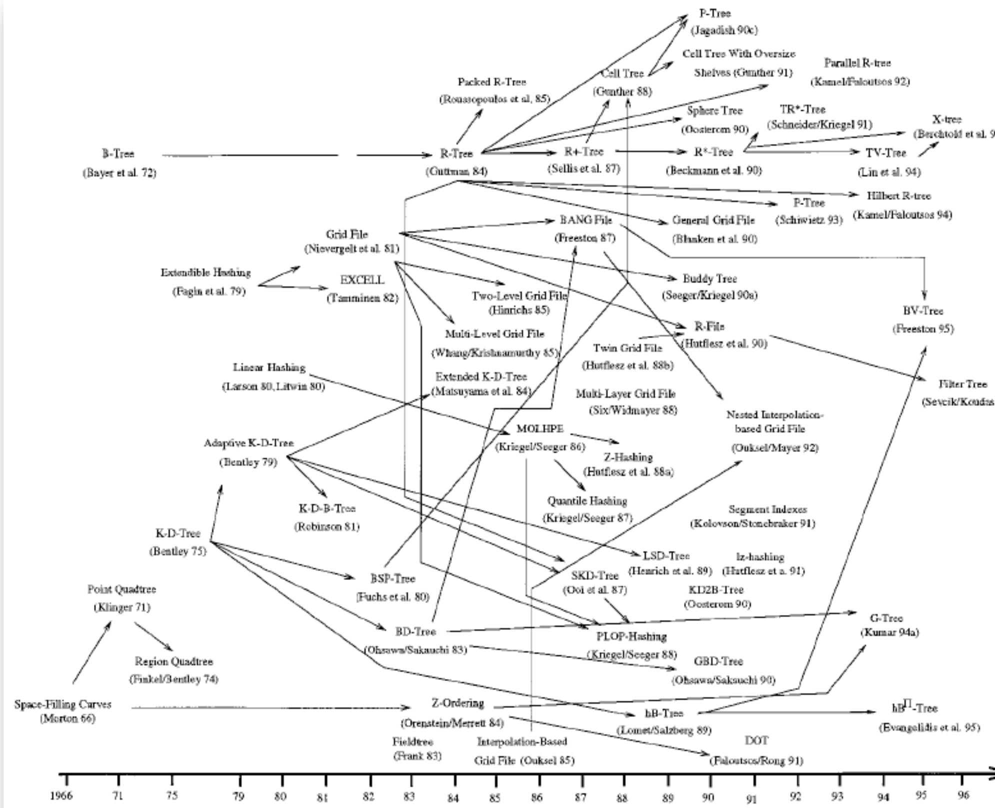


Figure 1. History of multidimensional access methods.

Next Topic: Why Indexing sometimes does not make sense for Multi-Dimensional Data.