

Lecture

# Database Implementation Techniques / Databases II

*OvGU Magdeburg, WinSem 2009/2010*

Sandro Schulze, Gunter Saake

{sanschul,saake}@iti.cs.uni-magdeburg.de.

# 5. Specific Access Structures

---

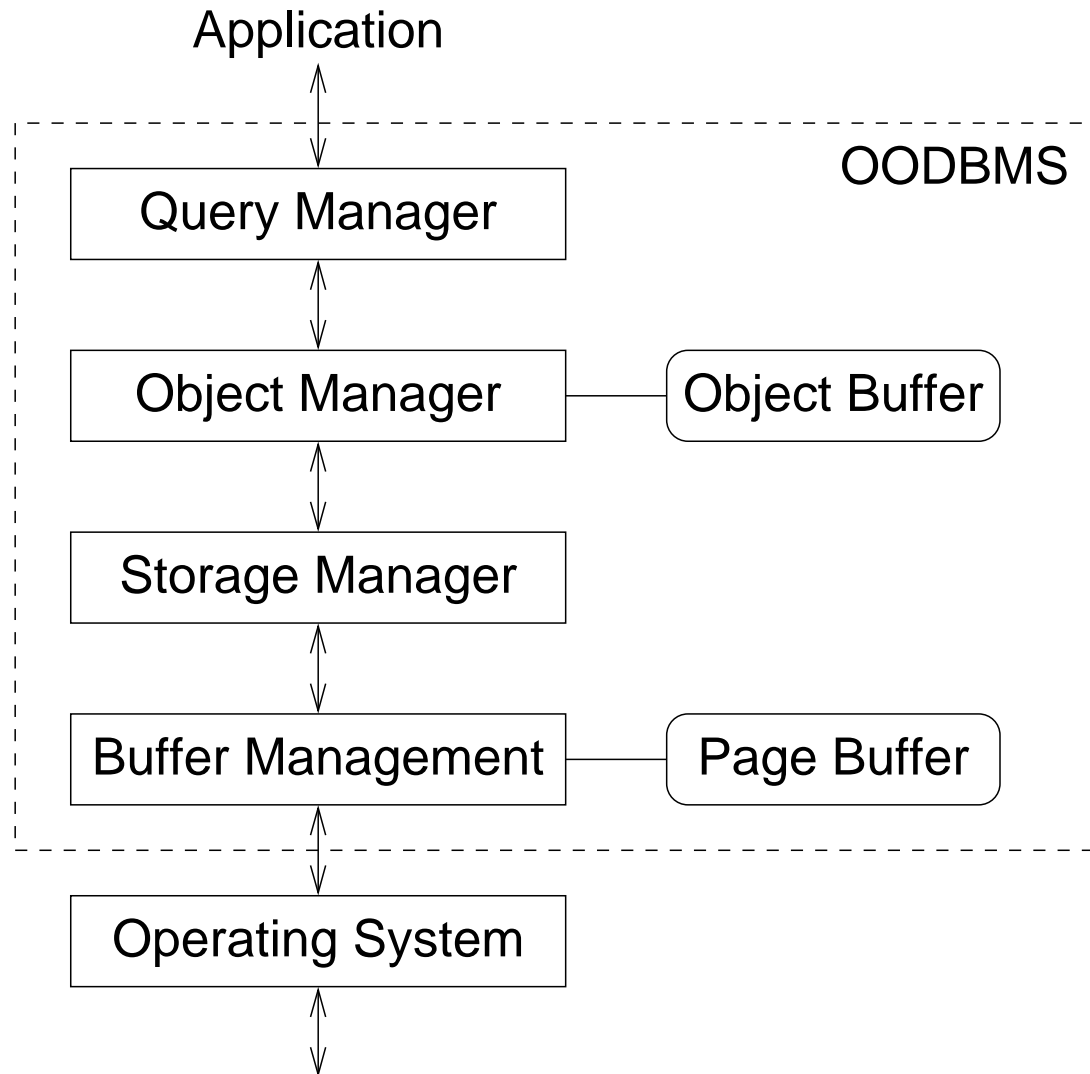
Access structures for specific applications

- Object-oriented databases
- Geometric access structures
- Access techniques for multimedia and text data

# Object-Oriented Databases

---

- Structure of an OODBMS



# Object Identity

---

- Variants
  - ◆ *Representation of object identity by surrogates and their implementation through indirect references: (GemStone, ORION/ITASCA, Postgres)*
  - ◆ *Representation and implementation of object identity through direct references: (ONTOS, ObjectStore)*
- First technique is (logical) better, but more slowly (because of indirections)

# Object Identity (II)

---

- Surrogates:
  - ◆ Always *unique*, even after deletion of the corresponding object
  - ◆ In *distributed environment* unique as well (encoding of computer ID in surrogate)
  - ◆ *Abstract class*, in which the object is instantiated, can be made explicitly within the surrogate
- Length: 32 or 64 Bit

# Classes

---

(without complex attribute values or component objects)

- *Binary storage*

Object is stored together with one attribute respectively as binary relation

- *Object structure with integrated schema*

Schema information integrated in storage structure of each object (e.g., in ORION/ITASCA)

- *Object structure with external schema*

# Classes (II)

---

## ■ Object structure with integrated schema

Surrogate	Bytes	Num_Attribute	Attribute	
011001010	38	3	workplace	...

Attribute		Werte-Offsets		
...	salary	supervisor	0	10

Attribute Values				
...	14	comp. science	3050	1001101

## ■ Object structure with external schema

Surrogate	Bytes	Attribute Values		
011001010	10	comp. science	3050	10001101

# Complex Attributes

---

- Principle
  - ◆ Object bigger than page  $\Rightarrow$  several pages in form of a B-Tree (e.g., EXODUS)
  - ◆ Set of objects of one class are clustered
- *Complex attributes*
  - ◆ *Divided storage* (normalized like in RDBS)  
Iris, OSCAR
  - ◆ Entire object structure is stored into one *Cluster*  
DASDBS
- *Private component objects*: Cluster possible
- *Shared component object*: Reference to (original) component object

# Cluster

---

- Cluster definition at time of
  - ◆ *Class definition (in the schema):*  $O_2$
  - ◆ *Object instantiation (per Object):* ObjectStore, ObServer, ONTOS and GemStone
- Cluster structures used for
  - ◆ all objects of a class
  - ◆ certain parts of classes, e.g., a partition of classes, based on certain attribute values
  - ◆ all instances of a class, which belong to a specified part of the class hierarchy
  - ◆ composed object (object with component objects)
  - ◆ complex attribute values

# Class Hierarchies

---

- Object exactly in one class:
  - ◆ State of the object is stored in this class (*Home Class Model*) (done in OODBPLs and some new developments, e.g., ORION)

# Class Hierarchies (II)

---

- Object in multiple classes
  - ◆ Object in smallest class (regarding the class hierarchy), together with inherited attribute values (*Leaf Overlap Model*) (in ORDBMS like Illustra)
  - ◆ Object in every class, together with local attribute values (*Split Instance Model*) (OpenODB)
  - ◆ Object in every class, together with attribute values defined there as well as inherited attribute values (*Repeat Class Model*) (deep extension, direct storage, e.g., UniSQL)
  - ◆ All objects in one file, non-applicable attributes are set to **null** (*Universal Class Model*)
  - ◆ All Objects in a ternary file with surrogate, attribute and attribute value (*Value Triple Model*)

# Access Paths for Classes

---

- Basic file organisation form already determined by storage structure of class
- Can be supported additionally by hash functions or B-Trees
- Support of access on objects in class and component hierarchies
- RDBMS: Access path supports only one relation
- OODBS: Support of set of classes by access path

# Access Paths for Class Hierarchies

---

- Index for hierarchy of classes on attribute of one (super)class  $C$ ; reference to
  - ◆ alle occurrences of suitable objects in class hierarchy with root  $C$ , if objects are stored using the split instance method
  - ◆ occurrence of object in class hierarchy in all other cases (i.e., methods), whereas the object can be stored in one of the subclasses of  $C$  as well
- *Class hierarchy index* (ORION/ITASCA,  $O_2$ )
- Example: Persons, employees and students: Index on Name of student

# Component Hierarchies

---

- Path expressions have to be supported; attribute values given for a (even indirect) component class

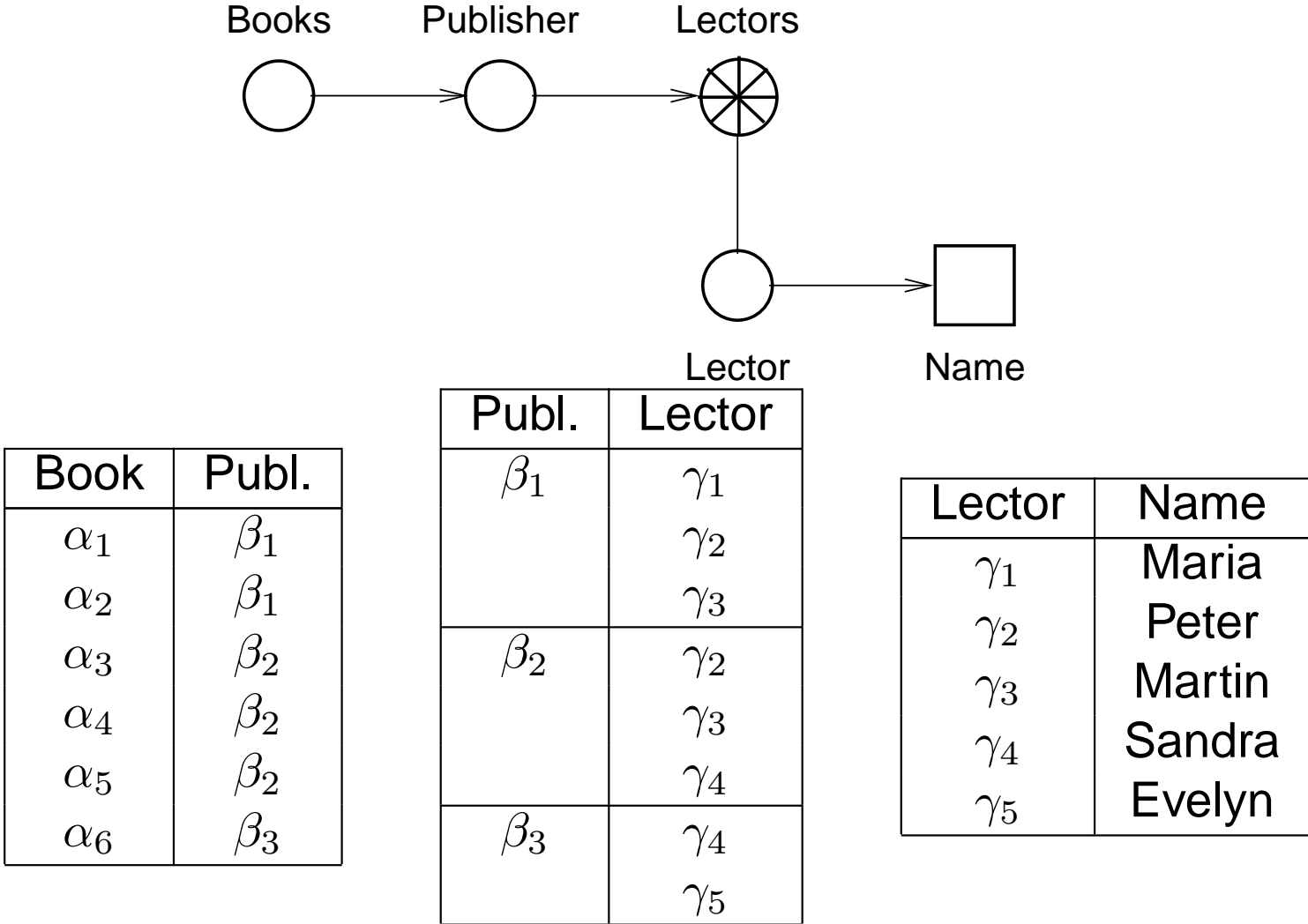
`Book.Publisher.PublisherLocation`

(Access on book over location of publisher)

`Book.Publisher.Lector.Name`

(Access on book over name of lector of the publisher)

# Component Hierarchies (II)



# equality and identity index

---

- *Support of attributes, whose type is a standard data type (e.g., String), analog to classical access paths in relational systems*

GemStone: *equality index* for sub-paths

`Publisher.PublisherLocation Or Lector.Name`

- *Support of component objects (i.e., object-value attributes)*

GemStone: *identity index* for sub-paths

`Book.Publisher Or Publisher.Lector`

# Types of Realization: Overview

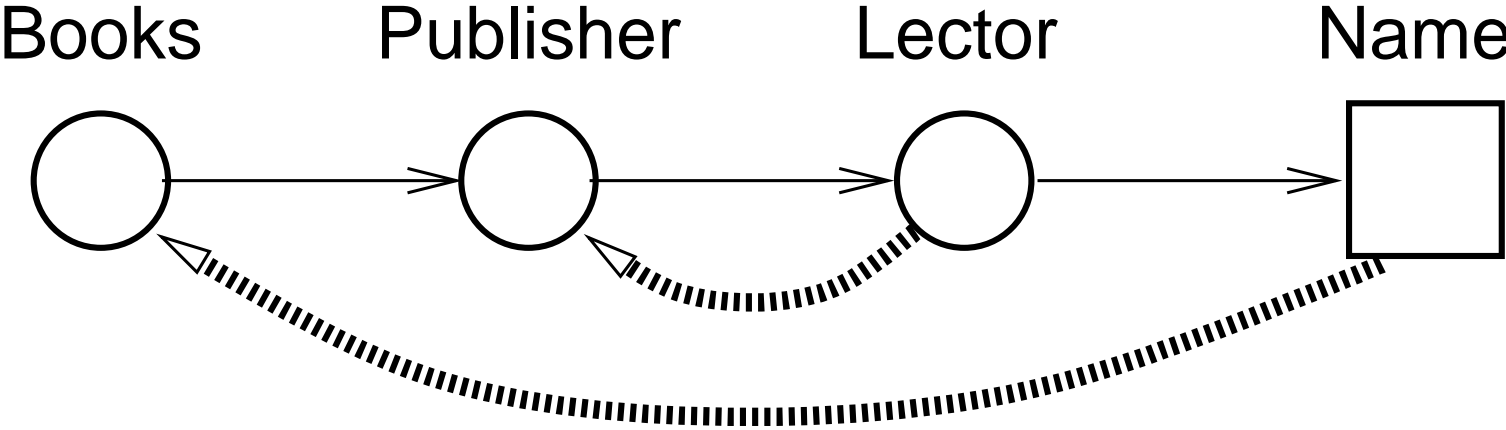
---

- *Path index*
- *Multi index*
- *Join index*
- *Nested index*
- *Access support relation*

(Type 1 and 2 in ORION and GemStone)

# Illustration: Index Graph

---



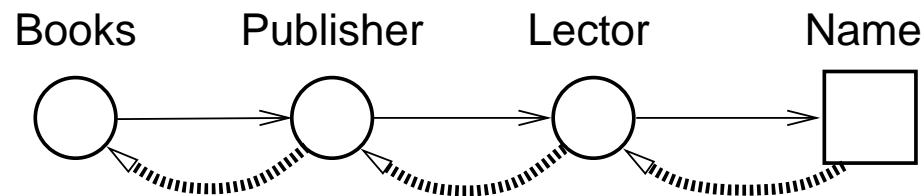
Legend:

—————>  
component relation

◁-----  
index support

# Multi Index

- *Multi index*: Binary index files, pointing from  $n$ -th component of path expression to  $n - 1$ -th component



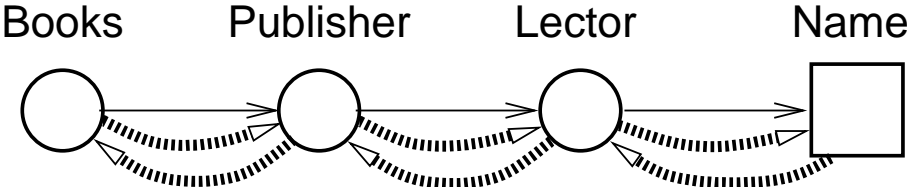
Name	Lector
Martin	$\{\gamma_3\}$
Eve	$\{\gamma_5\}$
Sandra	$\{\gamma_4\}$
Peter	$\{\gamma_2\}$
Maria	$\{\gamma_1\}$

Lector	Publ.
$\gamma_1$	$\{\beta_1\}$
$\gamma_2$	$\{\beta_1, \beta_2\}$
$\gamma_3$	$\{\beta_1, \beta_2\}$
$\gamma_4$	$\{\beta_2, \beta_3\}$
$\gamma_5$	$\{\beta_3\}$

Publ.	Book
$\beta_1$	$\{\alpha_1, \alpha_2\}$
$\beta_2$	$\{\alpha_3, \alpha_4, \alpha_5\}$
$\beta_3$	$\{\alpha_6\}$

# Join Index

- Join index: symmetric multi index



Name	Lector	Lector	Name	Lector	Publisher
Peter	$\gamma_3$	$\gamma_1$	Maria	$\gamma_1$	$\{\beta_1\}$
Eve	$\gamma_5$	$\gamma_2$	Martin	$\gamma_2$	$\{\beta_1, \beta_2\}$
Sandra	$\gamma_4$	$\gamma_3$	Peter	$\gamma_3$	$\{\beta_1, \beta_2\}$
Martin	$\gamma_2$	$\gamma_4$	Sandra	$\gamma_4$	$\{\beta_2, \beta_3\}$
Maria	$\gamma_1$	$\gamma_5$	Eve	$\gamma_5$	$\{\beta_3\}$

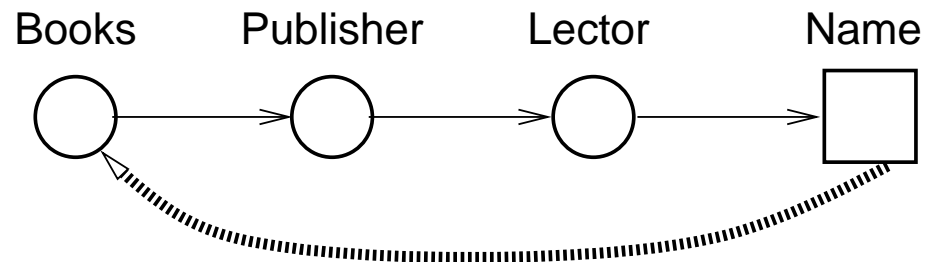
Publisher	Lector
$\beta_1$	$\{\gamma_1, \gamma_2, \gamma_3\}$
$\beta_2$	$\{\gamma_2, \gamma_3, \gamma_4\}$
$\beta_3$	$\{\gamma_4, \gamma_5\}$

Publisher	Book
$\beta_1$	$\{\alpha_1, \alpha_2\}$
$\beta_2$	$\{\alpha_3, \alpha_4, \alpha_5\}$
$\beta_3$	$\{\alpha_6\}$

Book	Publisher
$\alpha_1$	$\{\beta_1\}$
$\alpha_2$	$\{\beta_1\}$
$\alpha_3$	$\{\beta_2\}$
$\alpha_4$	$\{\beta_2\}$
$\alpha_5$	$\{\beta_2\}$
$\alpha_6$	$\{\beta_3\}$

# Nested Index

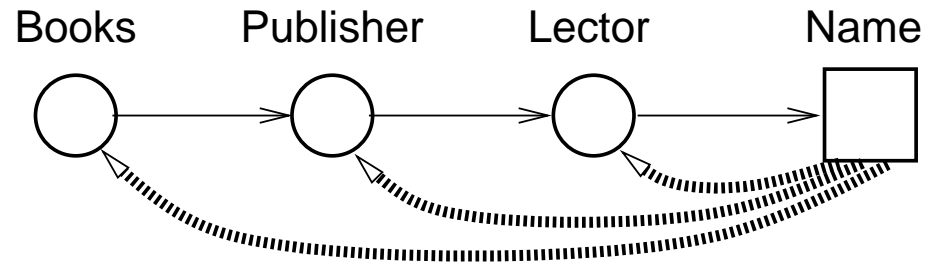
- *Nested index*: One single index file for  $n$ -th and first component of path expression



Name	Book
Martin	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Eve	$\{\alpha_6\}$
Sandra	$\{\alpha_3, \alpha_4, \alpha_5, \alpha_6\}$
Peter	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Maria	$\{\alpha_1, \alpha_2\}$

# Path Index

- *Path index*: Generalized nested index



Name	Lector
Martin	$\{\gamma_3\}$
Eve	$\{\gamma_5\}$
Sandra	$\{\gamma_4\}$
Peter	$\{\gamma_2\}$
Maria	$\{\gamma_1\}$

Name	Publisher
Martin	$\{\beta_1, \beta_2\}$
Eve	$\{\beta_3\}$
Sandra	$\{\beta_2, \beta_3\}$
Peter	$\{\beta_1, \beta_2\}$
Maria	$\{\beta_1\}$

Name	Book
Martin	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Eve	$\{\alpha_6\}$
Sandra	$\{\alpha_3, \alpha_4, \alpha_5, \alpha_6\}$
Peter	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Maria	$\{\alpha_1, \alpha_2\}$

# Access Support Relation (ASR)

---

- Generalization of all previous access paths, e.g., generalized (compact) path index

Name	Lector	Publisher	Book
Martin	$\{\gamma_3\}$	$\{\beta_1, \beta_2\}$	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Eve	$\{\gamma_5\}$	$\{\beta_3\}$	$\{\alpha_6\}$
Sandra	$\{\gamma_4\}$	$\{\beta_2, \beta_3\}$	$\{\alpha_3, \alpha_4, \alpha_5, \alpha_6\}$
Peter	$\{\gamma_2\}$	$\{\beta_1, \beta_2\}$	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Maria	$\{\gamma_1\}$	$\{\beta_1\}$	$\{\alpha_1, \alpha_2\}$

# Access Paths for Methods

---

- Results of method execution are stored in index
  - ◆ Non-parametrized method: one method result per object in index
  - ◆ Parametrized method: method result is stored per object and per possible parameter value in index; not efficient, hence, only certain ranges from value set in index  
or only parameter in index, already used for a query  
(*adaptive, learning index*)
- Materialization of method results:  
*Function-Materialization technique*

# Object Buffer

---

- So far:
  - ◆ Application data are loaded from page buffer to main memory parts, available for application programs
  - ◆ This „costs“ one transformation from internal representation to the representation desired by the application program
- In some systems, objects are loaded directly from disk to application storage: (O<sub>2</sub>, Objectivity, ONTOS), probably with certain address transformations (ObjectStore, see „Pointer Swizzling“)
- Other OODBs like GemStone, ORION/ITASCA, DASDBS and OSCAR have second buffer: *Object buffer*

# Pointer Swizzling

---

- Task: An object, which is present in main memory and accessed by an application program, should be found very fast
- with object buffer and logical object identities:
  1. Object  $\alpha$  in main memory possesses a component object  $\beta$ , which can not be found in the object buffer
  2. Object  $\beta$  is sought-after in page buffer
    - ◆ Search through of a *Resident Object Table (ROT)*
    - ◆ Assumption:  $\beta$  is not in page buffer

# Pointer Swizzling (II)

---

- with object buffer and logical object identities (cont.):
    3. (Re)load  $\beta$  from secondary storage
      - ◆ Search in *Persistent Object Table (POT)* to determine the secondary storage address for the given object identity
    4. After loading the object into the page or object buffer respectively: For every access (on object) in application program, the corresponding main memory address has to be searched using the logical object identity
- ~> to cumbersome and indirect

# Pointer Swizzling (III)

---

- In the case that direct references for the implementation of the object identity are used
  - ◆ Idirection is omitted
  - ◆ Nevertheless: Direct secondary storage address not useful in main memory  $\rightsquigarrow$  has to be transformed for every access.
- If object buffer is omitted:
  - ◆ No transformation from page buffer needed
  - ◆ However, objects contained in the page buffer have to be searched using the current main memory address as well

# Pointer Swizzling (IV)

---

- Hence: Transformation of indirect or direct (secondary storage-)references to main memory addresses  $\rightsquigarrow$  *Pointer Swizzling*
- Variants:
  - ◆ Original or copy:
    - Pointer transformation to original page (in page buffer) or to the copy (in object buffer)
  - ◆ Immediately or delayed:
    - Pointer of all objects are transformed while they are loaded or delayed with the first access on the object in main memory

# Pointer Swizzling (V)

---

- Variants(cont.):
  - ◆ Direct or indirect:
    - Transformation to direct main memory address or „only“ to a descriptor (indirect pointer), which contains main memory address
- Systems
  - ◆ ORION: immediate, indirect Swizzling
  - ◆ O<sub>2</sub>: no Pointer Swizzling
  - ◆ Exodus: delayed strategy
  - ◆ ObjectStore: immediate, direct Swizzling (VMMA)

# Geometric Access Structures

---

- Huge amount of geometric objects ( $> 10^6$ )
- Characteristics of geometric objects (geo objects)
  - ◆ Geometry (e.g., frequency polygon)
  - ◆ For support of queries: Additionally, a  $d$ -dimensional, describing rectangle (*bounding box*) can be used
  - ◆ Non-geometric attributes
- Application szenarios: Geo information systems (land register data, maps), CAx applications (e.g., VLSI design), ...
- Access is primarily realized over geo data: Search window (screen capture of data), access on adjacent objects

# Typical Operations

---

- Exact search
  - ◆ Input: Exact geometric search data
  - ◆ Result: One, single object (maximum)
- Range query for given,  $n$ -dimensional window
  - ◆ Search window:  $d$ -dimensional rectangle (corresponds to a multidimensional interval)
  - ◆ Result: All geo objects, which subtend the search window
  - ◆ Result quantity depends on parameter (of search window)

# Typical Operations (II)

---

- Insertion of geo objects
  - ◆ *Desireable without global reorganisation!*
- Deletion of geo objects
  - ◆ *Desireable without global reorganisation!*

# Adjacency-preservative Search Trees

---

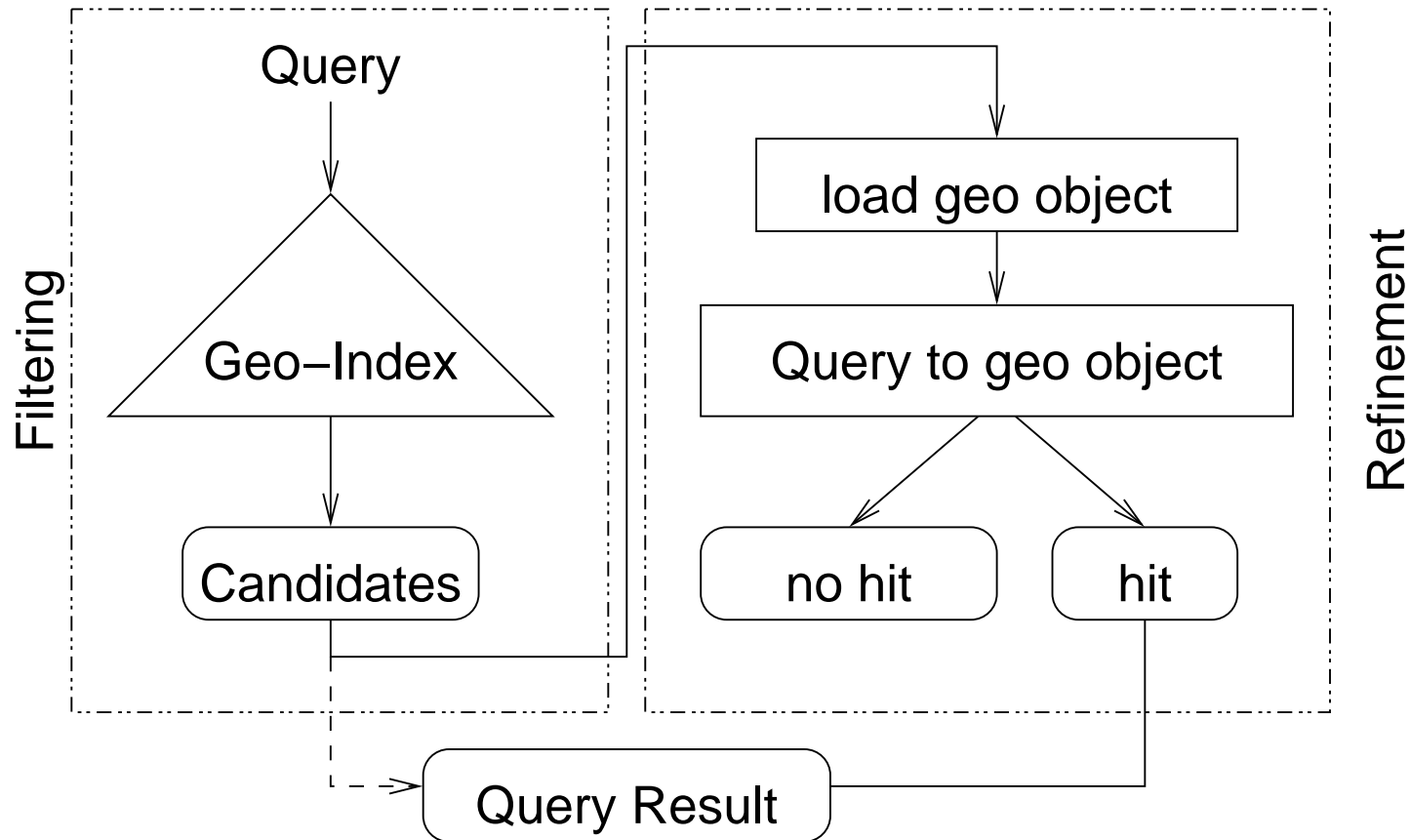
- Partition of geometric space in *regions*
- Adjacent objects should be assigned to *same* region, if possible
- If not applicable, distribute these objects on *adjacent* regions
- tree structure is created by refinement of regions to adjacent partial regions
- Storage of objects in leaf regions

# Adjacency-preservative Search Trees (II)

---

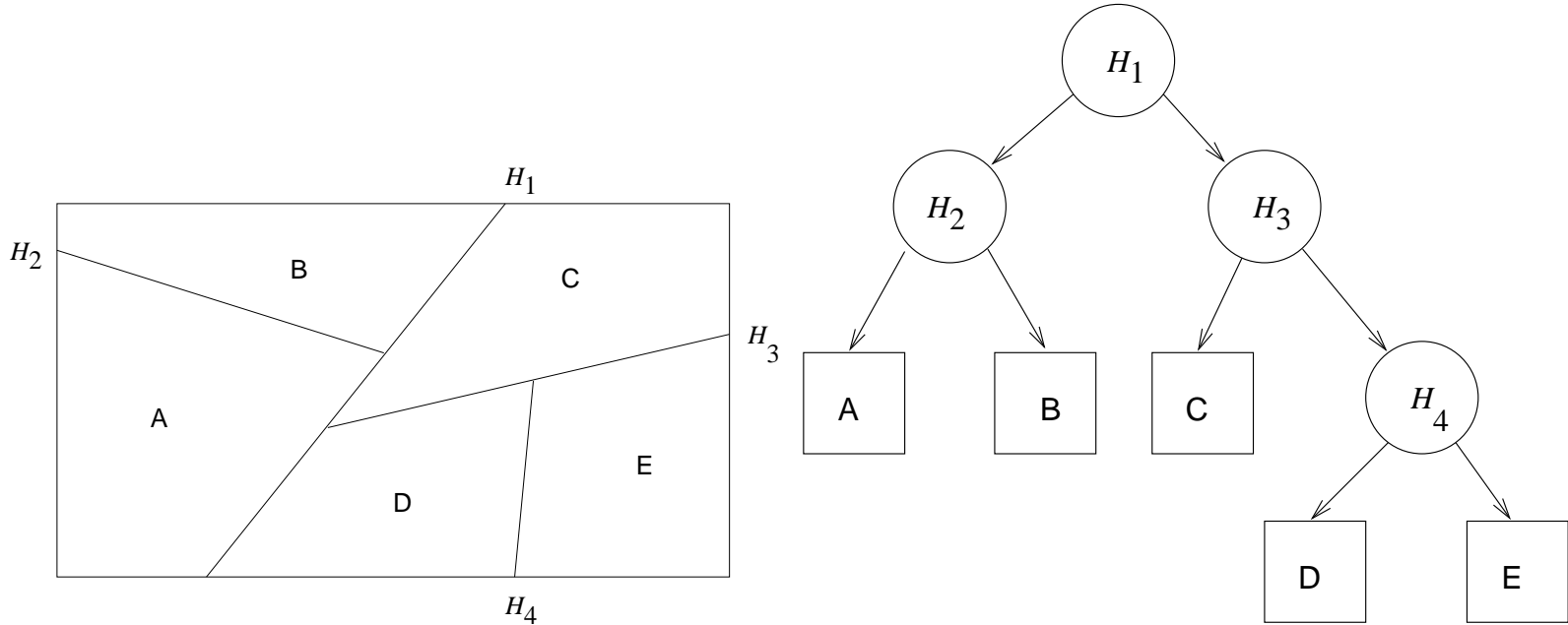
- Degrees of freedom
  - ◆ Shape of regions
  - ◆ Disjunctive (total) partitioning or overlapping of regions
  - ◆ Unique or multiple assignment of objects to regions
  - ◆ Storage and access using original or derived geometry for objects
  - ◆ Degree of tree & organisation form

# Multi-Level Processing of geom. Queries



# Geometric Tree Structure: BSP-Tree

---



# Realization Variants

---

Alternative	Tree Structures			
	BSP-Tree	R-Tree	R <sup>+</sup> -Tree	Cell Tree
Region shape	convex polygons	rectangles	rectangles	convex polygons
Part. regions	complete	incomplete	incomplete	complete
Overlapping	no	yes	no	no
Balanced	no	yes	yes	yes

# R-Trees (I)

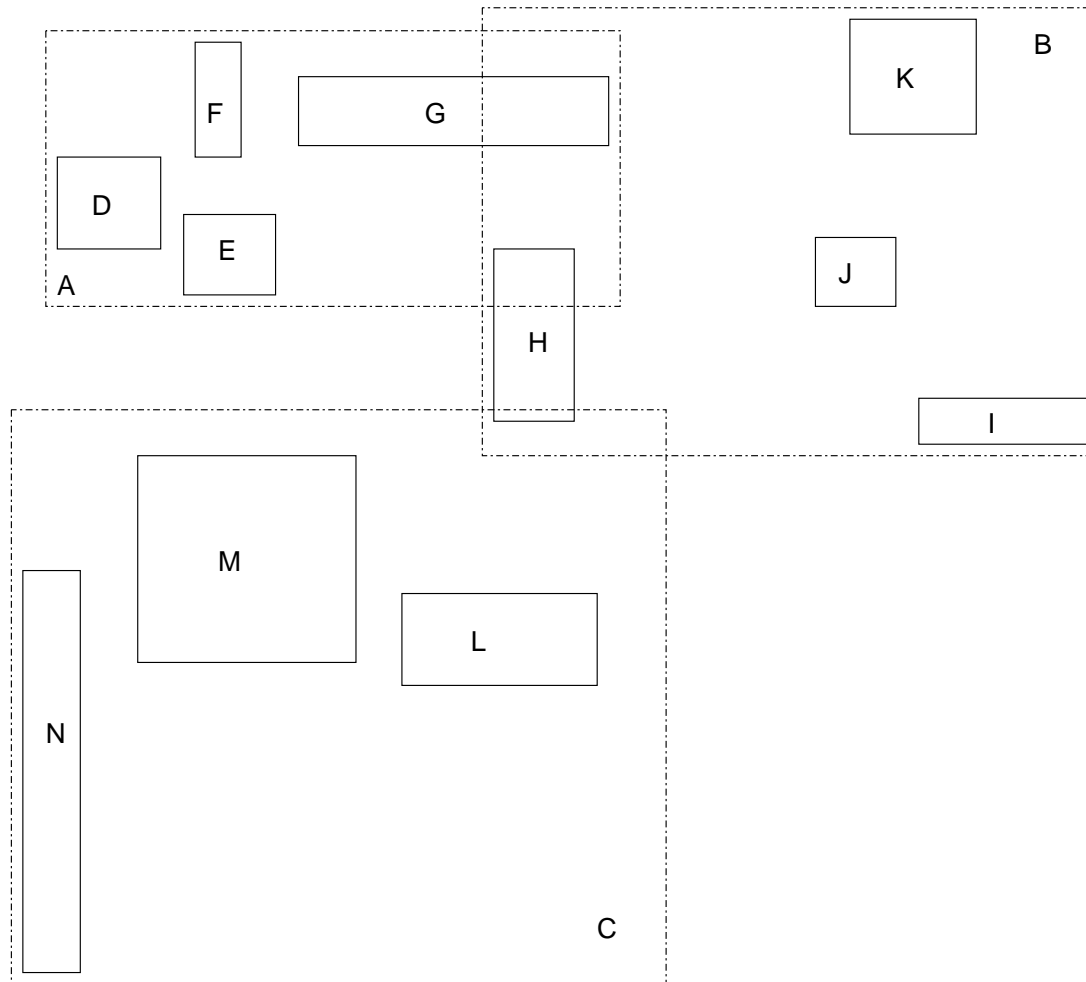
---

- R-Tree: Generalization of B-Tree principle to several dimensions
  - ◆ Root of tree is a rectangle, which encompasses all geo objects
  - ◆ Geo objects (itself) are represented by their enclosing rectangles
  - ◆ Partition in regions takes place using non-disjunctive rectangles
  - ◆ Each geo object is assigned unique to one leaf

# R-Trees (II)

---

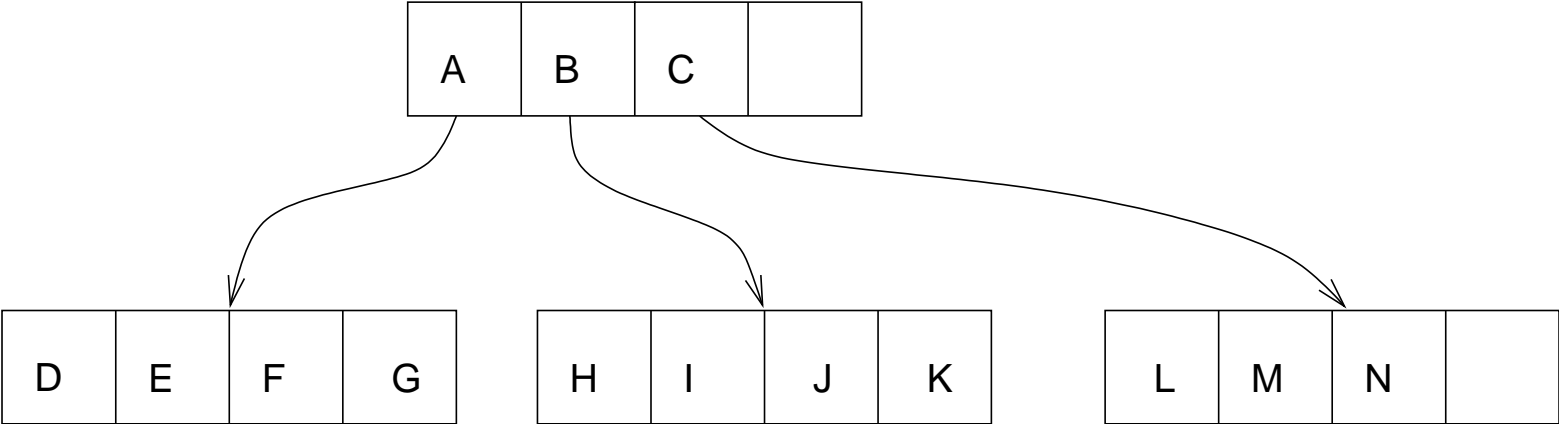
- Region partition through rectangles in the R-Tree



# R-Trees (III)

---

- Tree structure for R-Tree



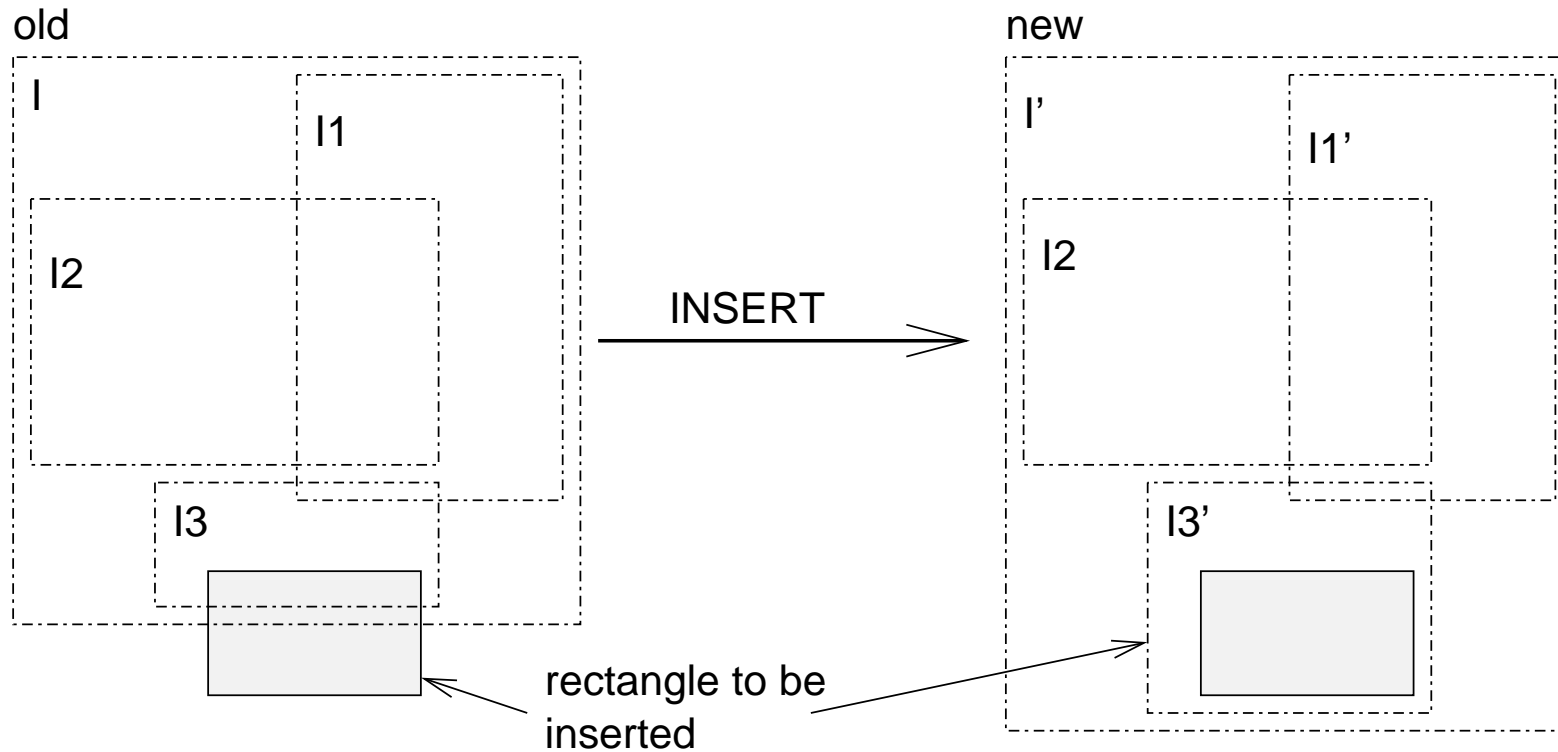
# Problems of R-Trees

---

- A certain rectangle can be overlapped by many region, but it is stored in exactly one region (only)
- Even point queries can result into a lookup of many rectangle regions
- Inefficient for exact match (which is also needed for insert and delete operations!)
- Problems with insertion
  - ◆ Insert requires often an expansion of regions (propagated upwards)

# Expansion in case of Insert

---



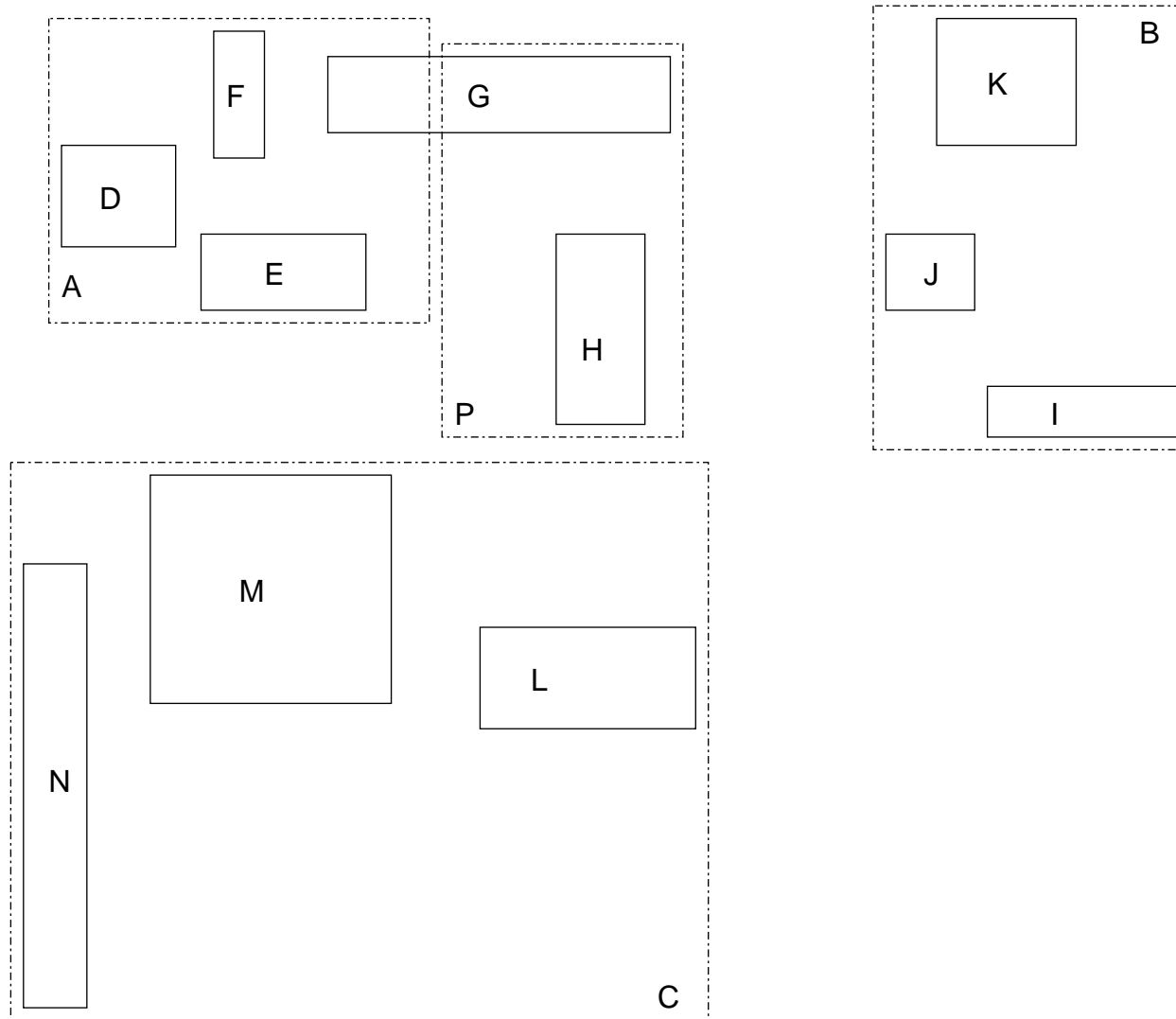
# R<sup>+</sup>-Trees (I)

---

- R<sup>+</sup>-Tree: Partition in *disjunctive* regions
- Hence, each stored point of a geometric object is assigned a unique leaf
- On each tree level, at most one rectangle is assigned to a point as well → distinct path from root to leaf (storing an point)
- ‘Clipping’ of geo objects necessary!

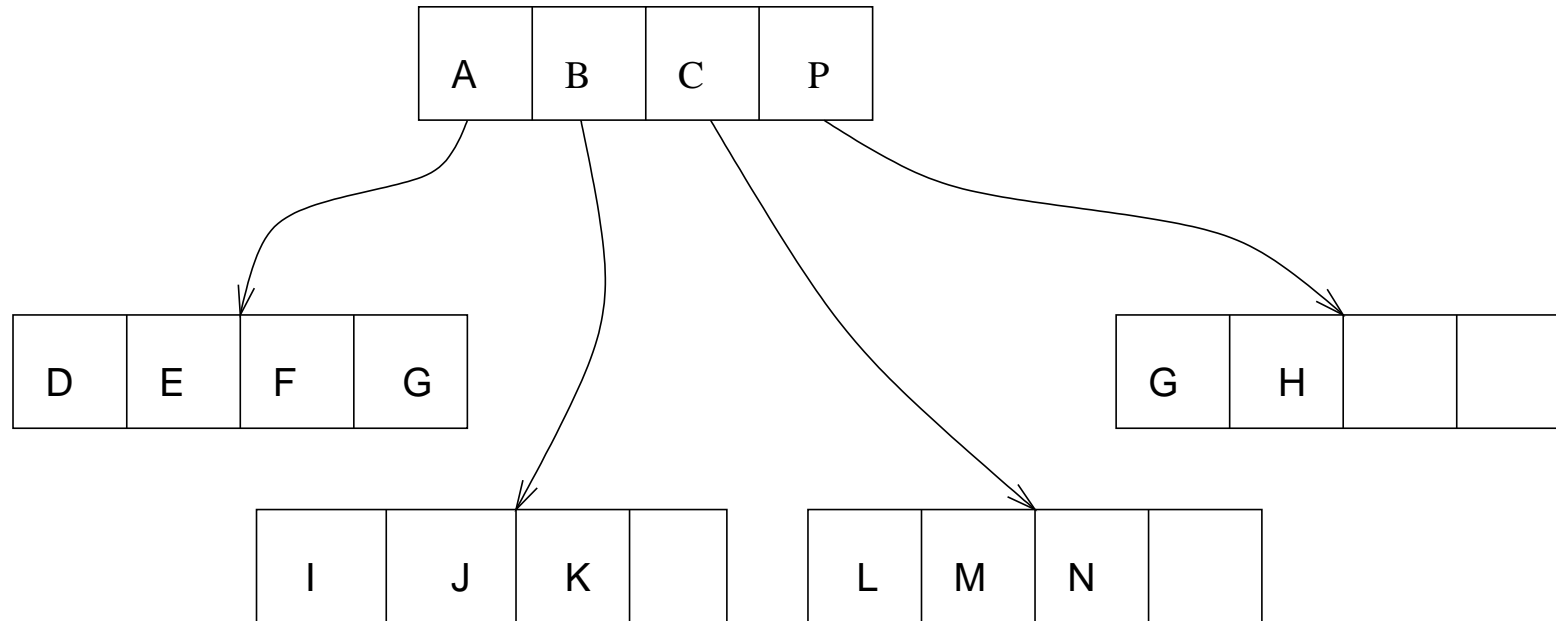
# R<sup>+</sup>-Trees (II)

---



# R<sup>+</sup>-Trees (III)

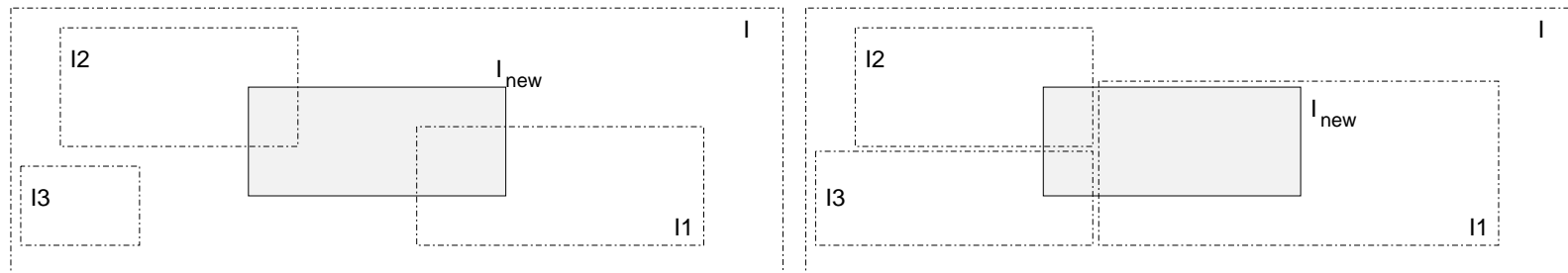
---



# Problems of $R^+$ -Trees

---

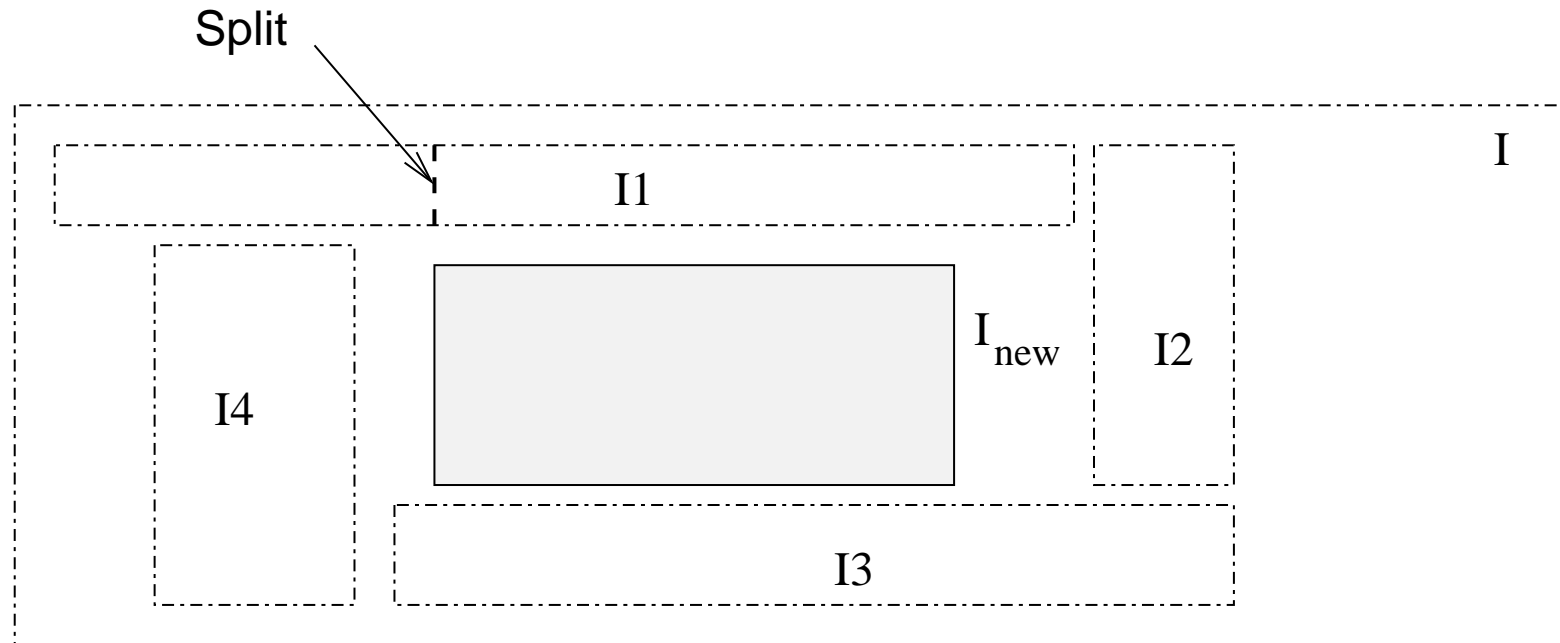
- Objects have to be stored in several rectangle regions (*clipping*) — increased storage & modification effort
- Insert of objects possibly requires modification of several rectangle regions



# Problems of $R^+$ -Trees (II)

---

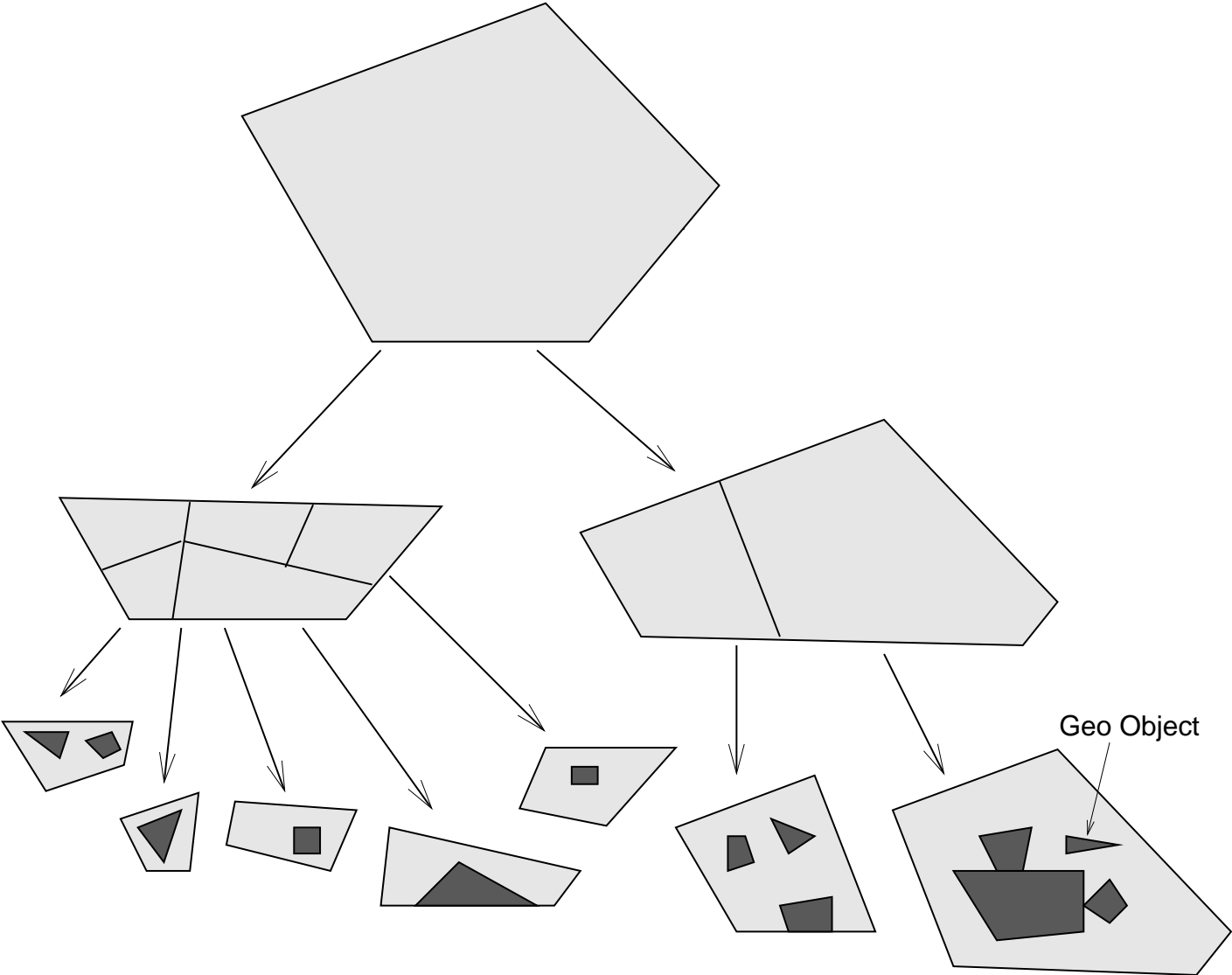
- Insert can lead to *unavoidable* partition of regions in certain situations



- Region modifications have consequences in both directions, to the leaves as well as to the root
- Upper bound for entries in leaf nodes can not be guaranteed anymore

# Cell-Trees

---



# Point Data Structures

---

- Storage of rectangle using point data structures (e.g., grid file)
  - ◆ Transformation of spacious objects (multidimensional rectangles) to point data
  - ◆ Transformation maps  $d$ -dimensional rectangles to points in  $2d$ -dimensional space  $\mathcal{R}^{2d}$
  - ◆  $d$ -dimensional rectangle(notations):

$$r = [l_1, r_1] \times \cdots \times [l_d, r_d]$$

# Point Data Structures (II)

---

- Corner transformation

$$p_r = (l_1, r_1, \dots, l_d, r_d) \in \mathcal{R}^{2d}$$

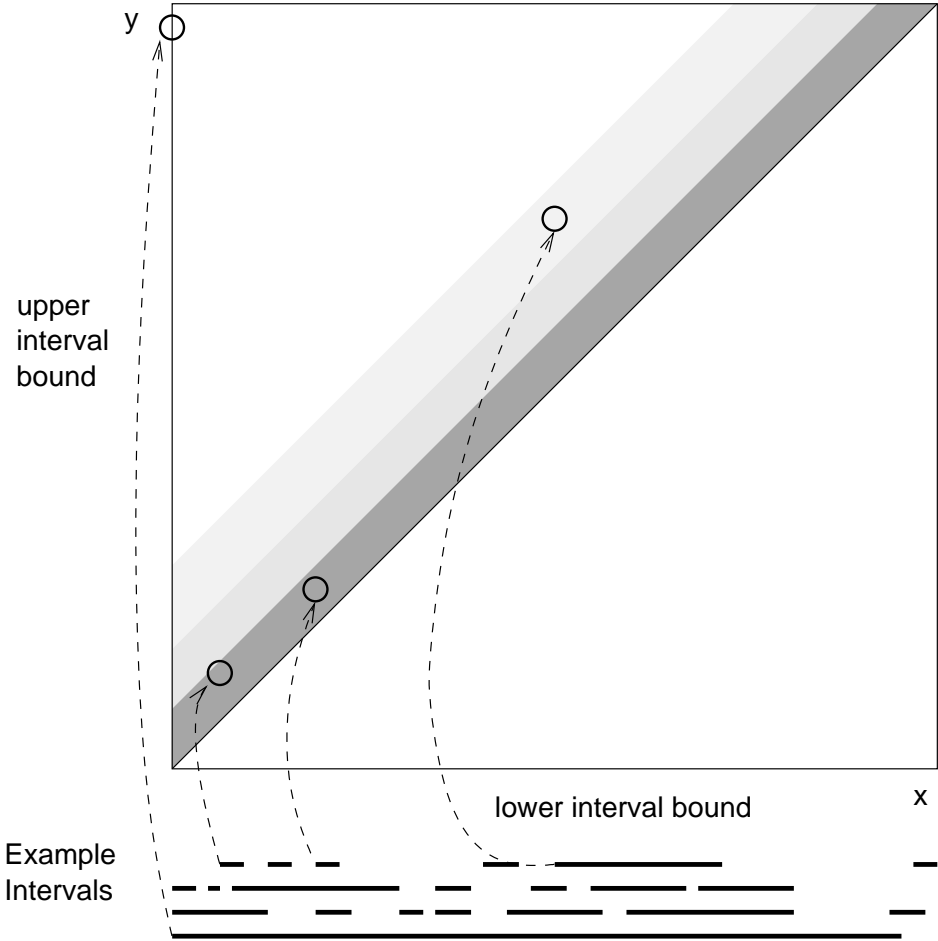
per interval as coordinates: upper bound, lower bound

- Center transformation

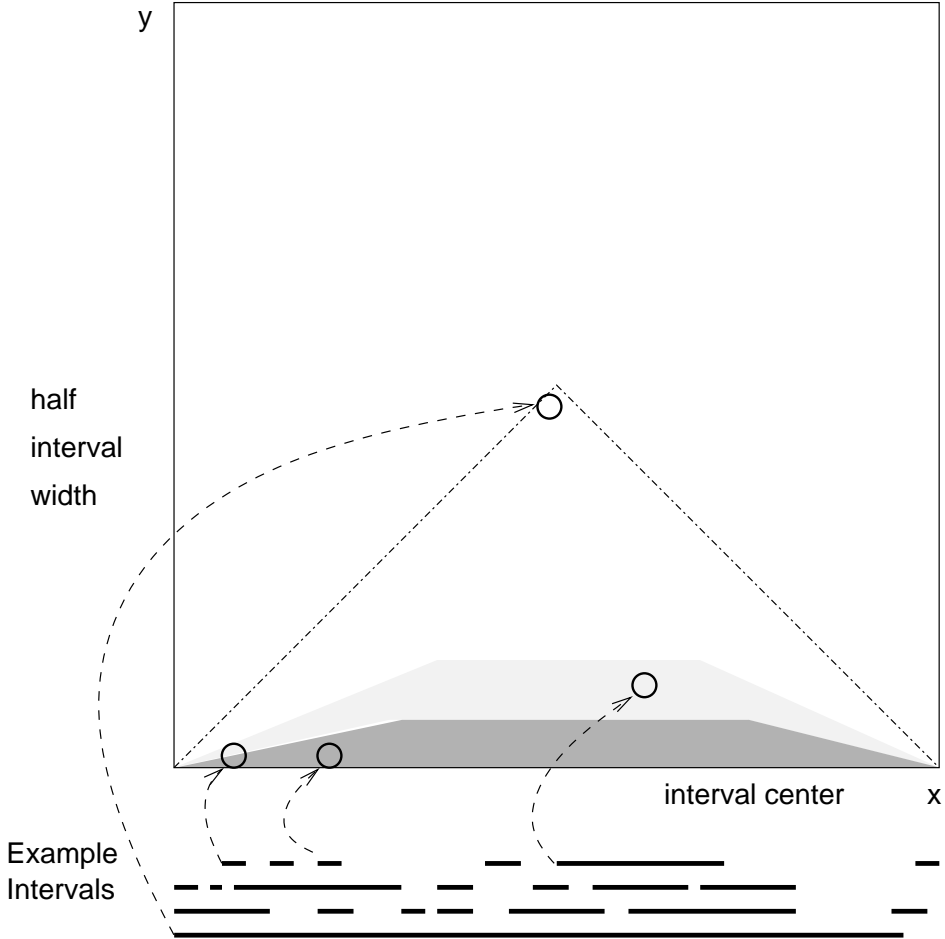
$$p_r = \left( \frac{l_1 + r_1}{2}, \frac{r_1 - l_1}{2}, \dots, \frac{l_d + r_d}{2}, \frac{r_d - l_d}{2} \right) \in \mathcal{R}^{2d}$$

per interval as coordinates: center point, half width

# Corner Transformation

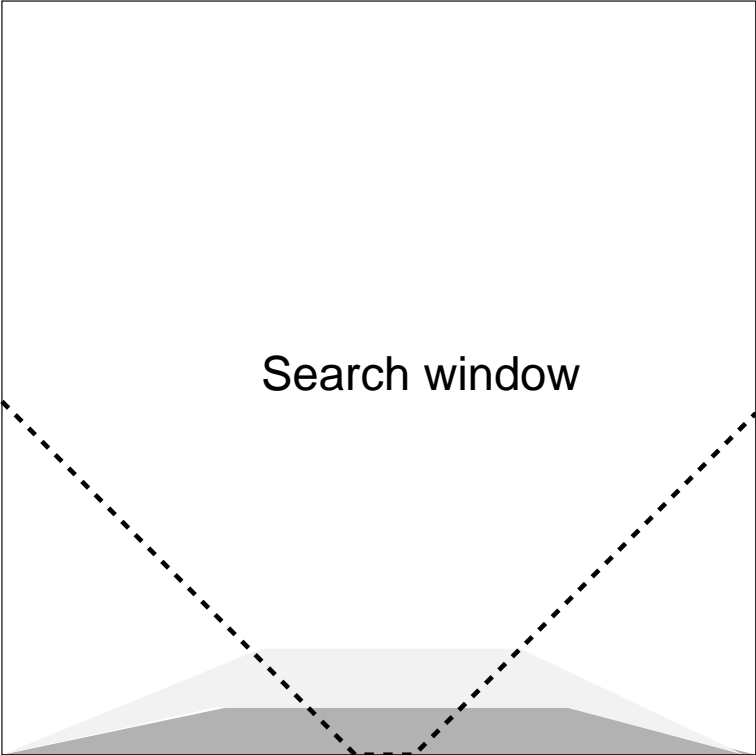
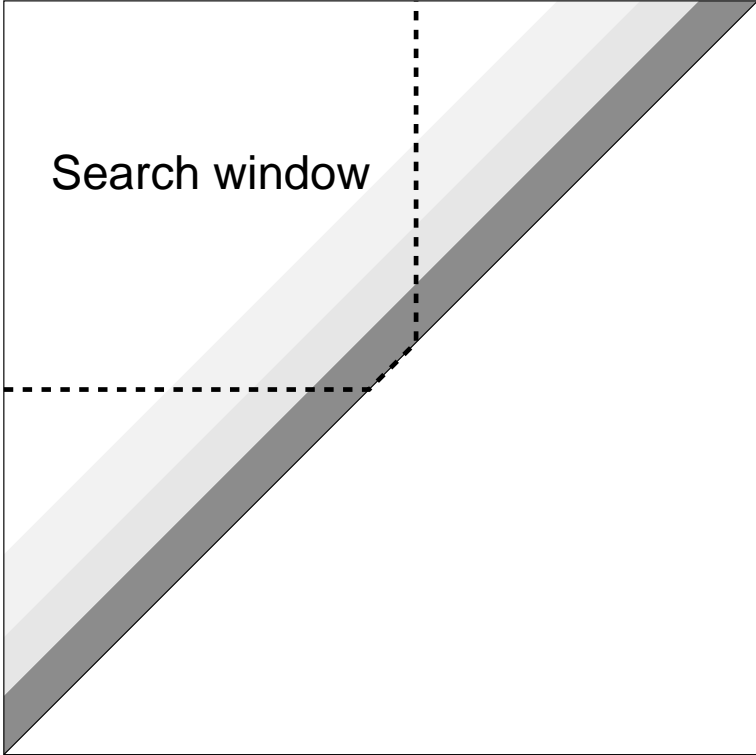


# Center Transformation



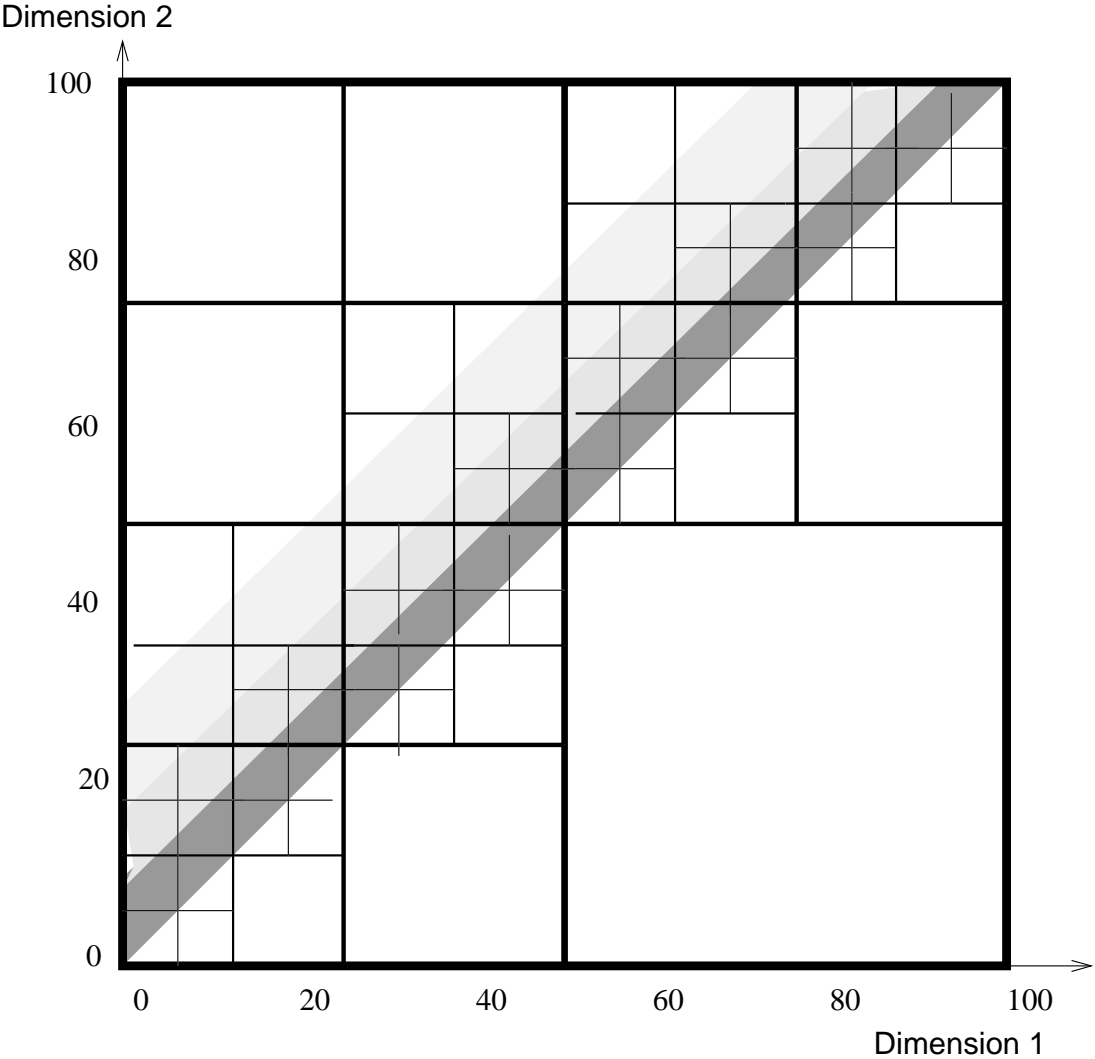
# Search Window

---



# Grid File Degeneration

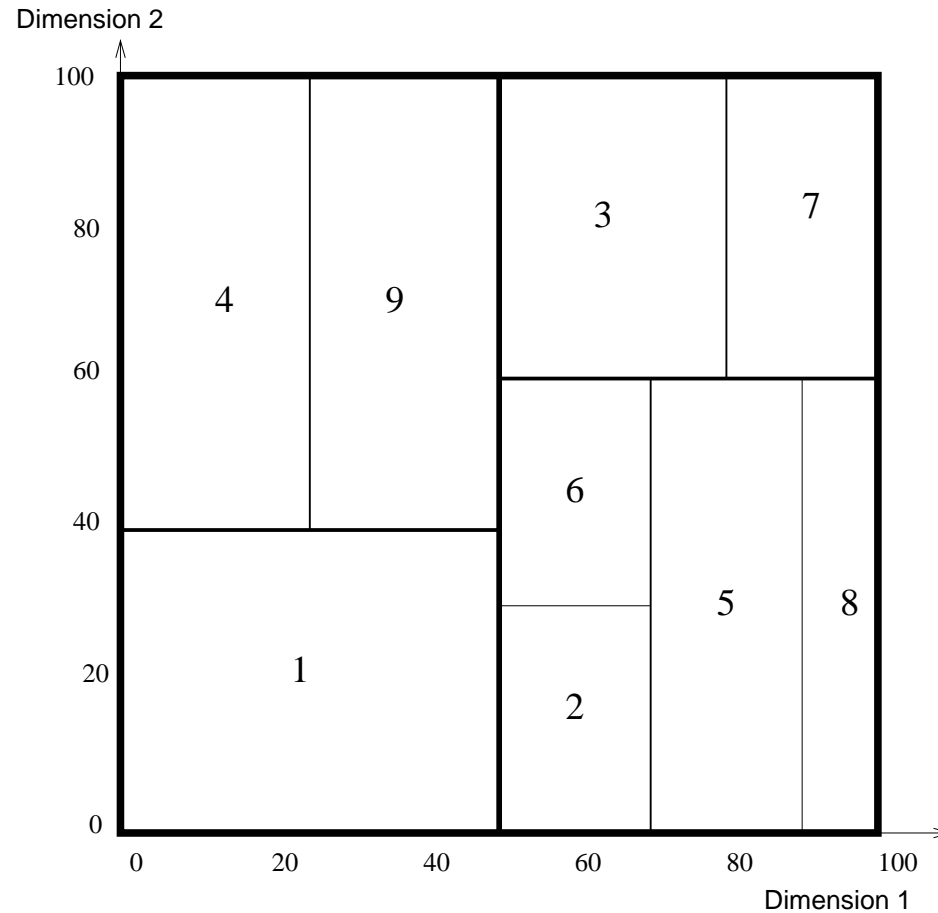
---



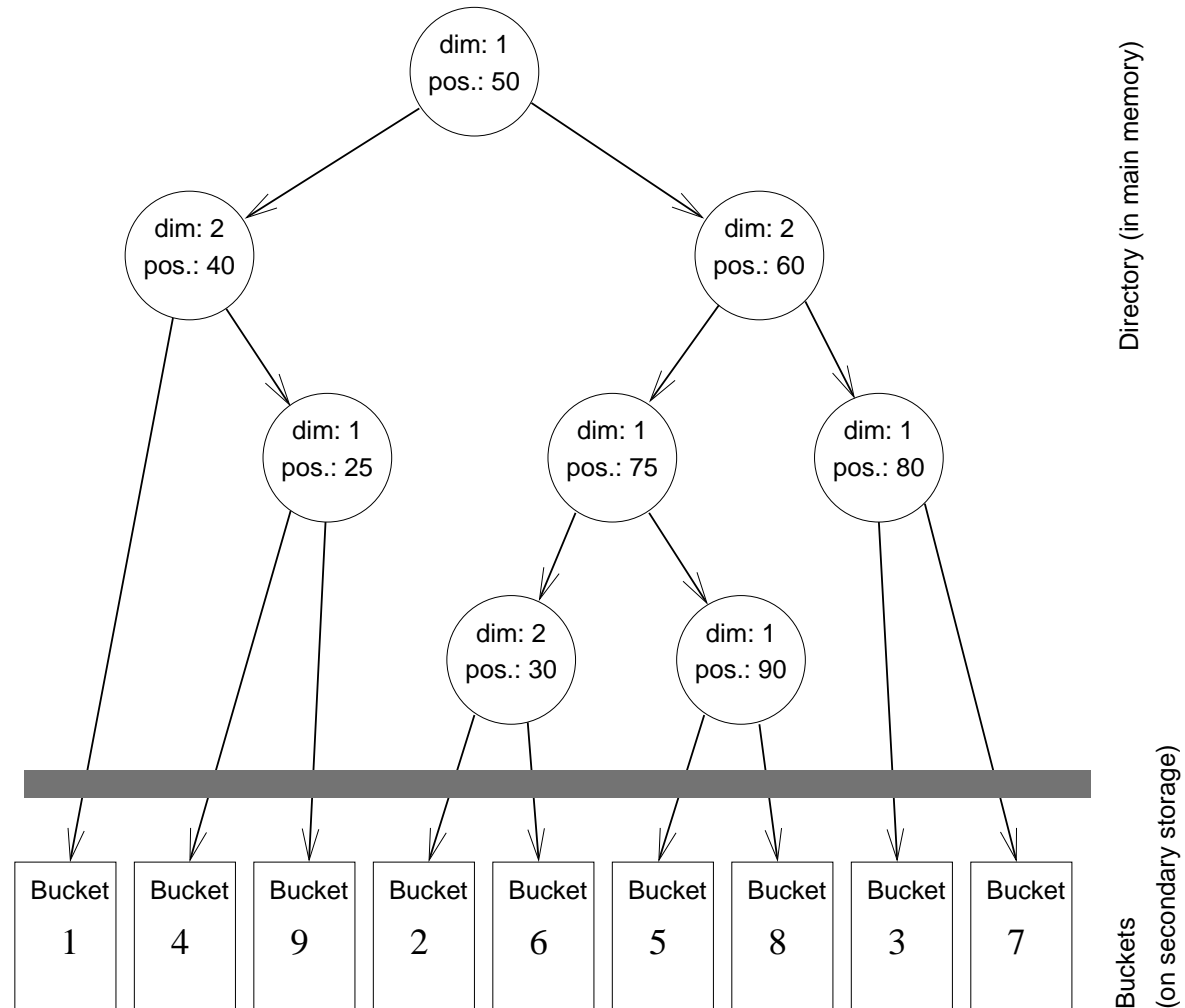
# Grid File Extensions: LSD-Trees

---

- Possible partition of one layer/plane for a LSD-Tree



# Example for LSD-Trees



# Access Structures for Multimedia DB

---

- Media objects → huge *binary objects*

Format	Megabyte	Compression
JPEG	75	uncompressed
MPEG-1	12.5	compressed, loss of quality
MPEG-2	17	compressed, high quality

1 minute of combined Audio/Video recording

- Indexing on derived attributes (so-called *Features*)  
→ Indexing on highdimensional *feature vectors*
- *Continuous data types*

# Continuous Data Types

---

- Data types for storage of *continuous media*
  - ◆ Data have to be loaded *fast enough* (in real-time) into main memory for playing.
  - ◆ Since bandwidth for (data) transfer from secondary storage to main memory is not constant, data are usually loaded *ahead into main memory cache* (*prefetching strategy*)
  - ◆ Caution: *Avoiding cache overflow*
  - ◆ Especially for continuous media: *Storage on tertiary storage medium* is common  $\leadsto$  two-level caching strategy necessary

# Highdimensional Indices

---

- Feature vectors for characterizing media objects
- Feature vector: Point in highdimensional space
- Database as set of points of the  $d$ -dimensional space:

$$DB = \{P_0, \dots, P_{n-1}\}$$

- Every point of this database is a  $d$ -dimensional vector:

$$P_i \in DB \subseteq \mathbb{R}^d$$

# Typical Operations on Feature Vectors

---

- *Range queries* compute all adjacent points in database for a given vector:

$$\mathbf{range}(DB, Q, r, M) = \{P \in DB \mid \delta_M(P, Q) \leq r\}$$

with:

- ◆  $DB$ : Database as set of feature vectors (search space)
- ◆  $r$ : Distance determining the range for searching
- ◆  $Q$ : Search vector
- ◆  $M$ : Metric (e.g., euclidean distance)

# Typical Operations on Feature Vectors (II)

---

- *Point query* defines the exact search and thus, corresponds to range query with distance 0:

$$\mathbf{point}(DB, Q, M) = \{P \in DB \mid \delta_M(P, Q) = 0\}$$

- *Nearest neighbor query* determines for a given (search) vector the nearest vector in the database:

$$\mathbf{nearest-neighbor}(DB, Q, M) =$$

$$\{P \in DB \mid \forall(P' \in DB) : \delta_M(P, Q) \leq \delta_M(P', Q)\}$$

# Distance Function $\delta$

---

1.  $\delta(p, p) = 0$
2.  $\delta(p_1, p_2) = \delta(p_2, p_1)$
3.  $\delta(p_1, p_2) + \delta(p_2, p_3) \geq \delta(p_1, p_3)$  (triangle inequality)

# Metrics

---

- *Euclidean distance*

$$\delta_{Euclid}(P, Q) = \sqrt{\sum_{i=1}^d (Q_i - P_i)^2}$$

$Q_i$  is value of  $Q$  for the  $i$ -th dimension

- *Manhattan distance*

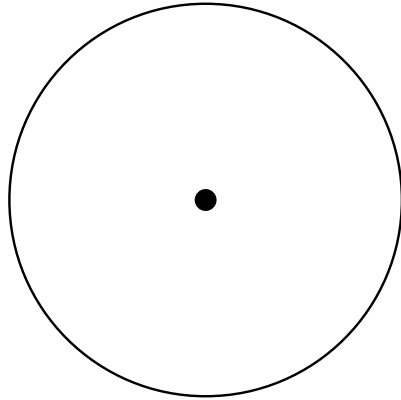
$$\delta_{Manhattan}(P, Q) = \sum_{i=1}^d |Q_i - P_i|$$

- *Maximum distance*

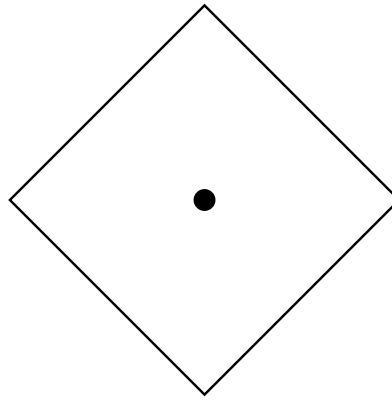
$$\delta_{Max}(P, Q) = \max\{|Q_i - P_i|\}$$

# Comparison of Metrics

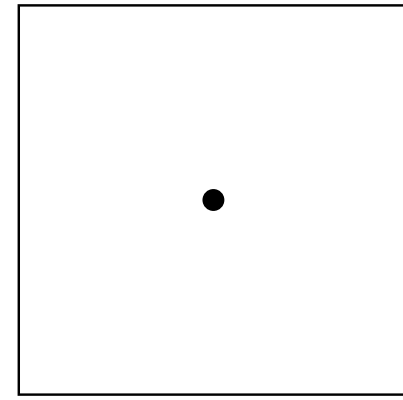
---



Euclid



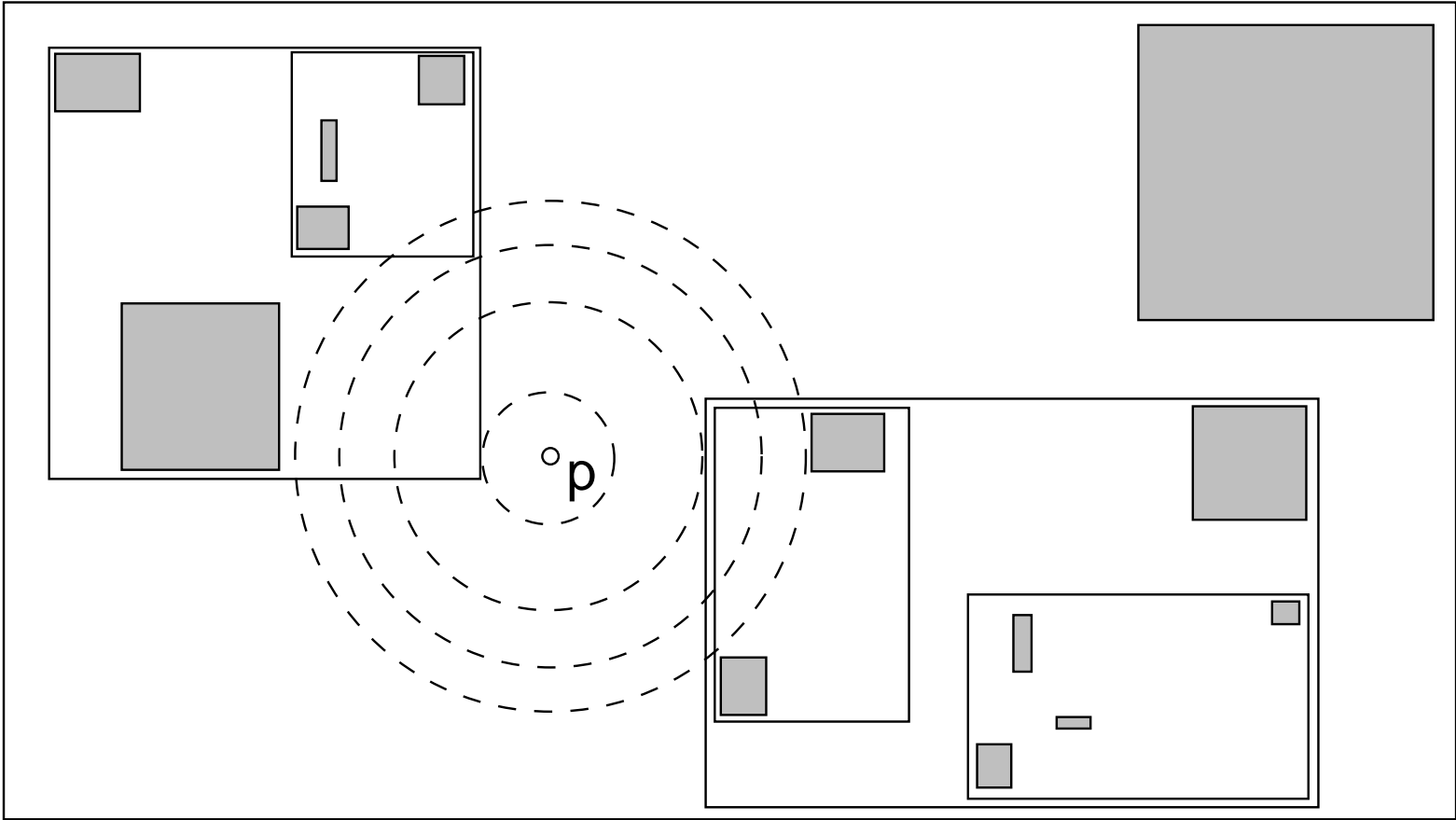
Manhattan



Max

# Nearest Neighbor Search in R-Trees

---



# MinDist and MinMaxDist

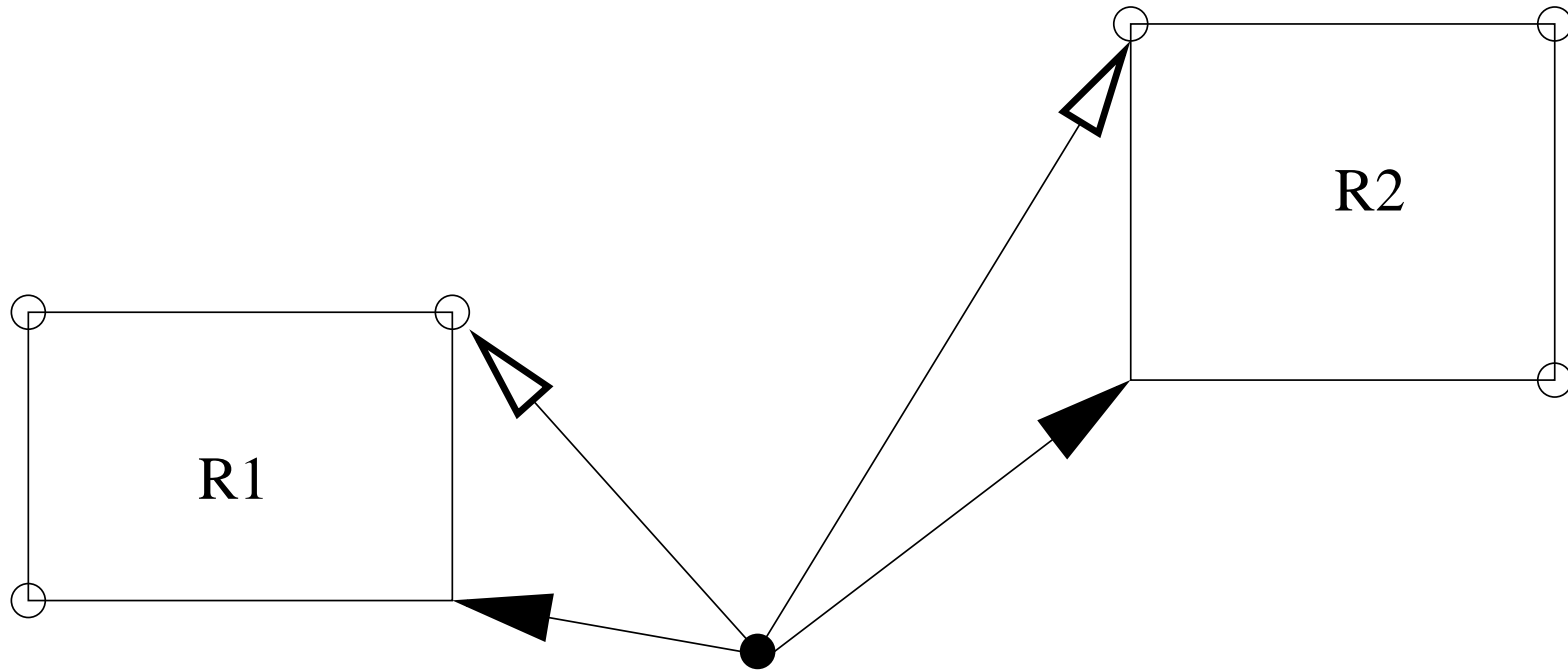
---

- **MinDist:** *Min-distance* is minimal distance between query point and MBR
- **MinMaxDist:** for every hyperplane ( $n - 1$  dimensions) the point, most far away from query point is determined (most disadvantageous case)
  - ◆ Min-Max point is then the minimal distant point of all of these max-points and its distance to query point is min-max distance

$$\text{MinDist} \leq \text{MinMaxDist}$$

# MinDist and MinMaxDist (II)

---



- MinDist
- ▷ MinMaxDist

# Nearest Neighbor

---

- Currently assumed minimal distance **Dist**
  - ◆ **Dist** < **MinDist** → Child nodes must not be sought, since they can not contain any point, which can be considered as hit
  - ◆ **MinDist** ≤ **Dist** ≤ **MinMaxDist** → Child nodes have to be sought, since a candidate could be contained
  - ◆ **Dist** > **MinMaxDist** → the considered MBR or a child node contain a candidate, which is closer than the point currently assumed to be the nearest one
  - ◆ Detection of closer candidate → Update of **Dist**

# X-Tree as an Example

---

- Performance problems of R-Tree structures with high number of dimensions:
  - ◆ Usually, large overlapping results from splitting of inner nodes, caused by selecting wrong dimensions (for the split)
  - ◆ For large overlapping, linear search is faster than block-oriented access on tree node

# X-Tree as an Example (II)

---

- X-Tree: Modifications of R-Trees
  - ◆ Constant factor **max\_overlap** determines a maximum degree of overlapping
  - ◆ For MBR, a *split history* containing dimensions for previous splits is maintained
  - ◆ *Super nodes* have the size of several normal, inner tree nodes and hence, can contain many entries

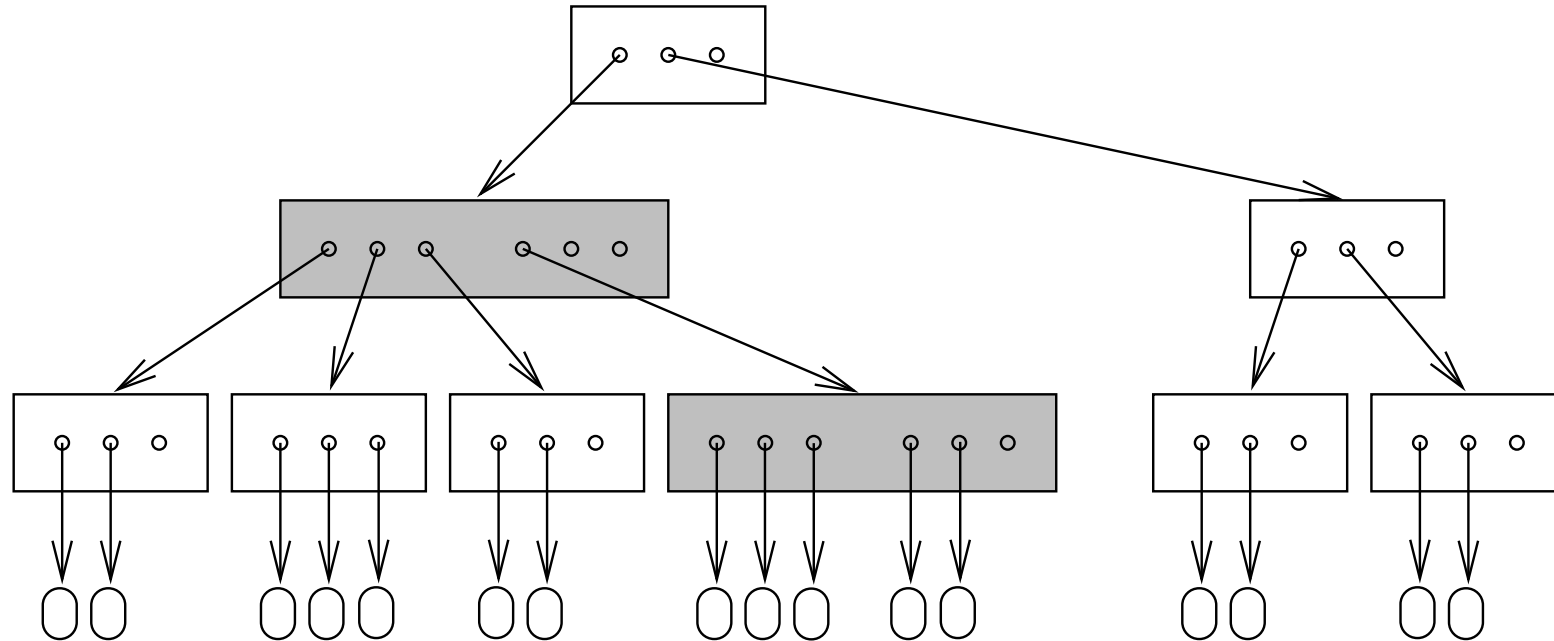
# X-Tree as an Example (III)

---

- Types of nodes:
  - ◆ Simple inner nodes: same structure as in R-Tree, but additionally split histories are stored
  - ◆ Inner *super nodes* span several data blocks and hence, reference to a higher number of entries (data)
  - ◆ Leaf nodes correspond to the leaves of the R-Tree

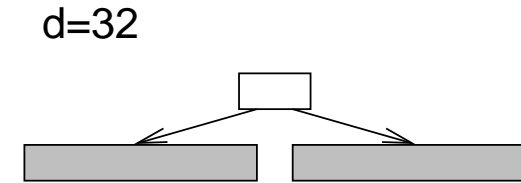
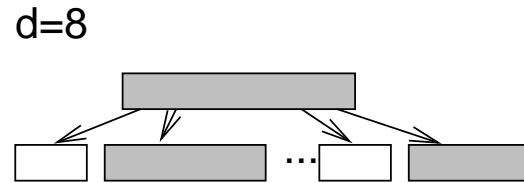
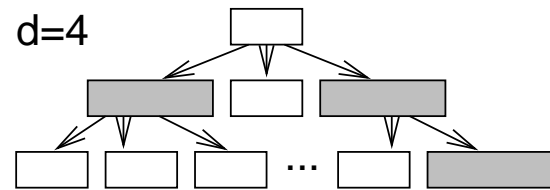
# X-Tree graphical

---



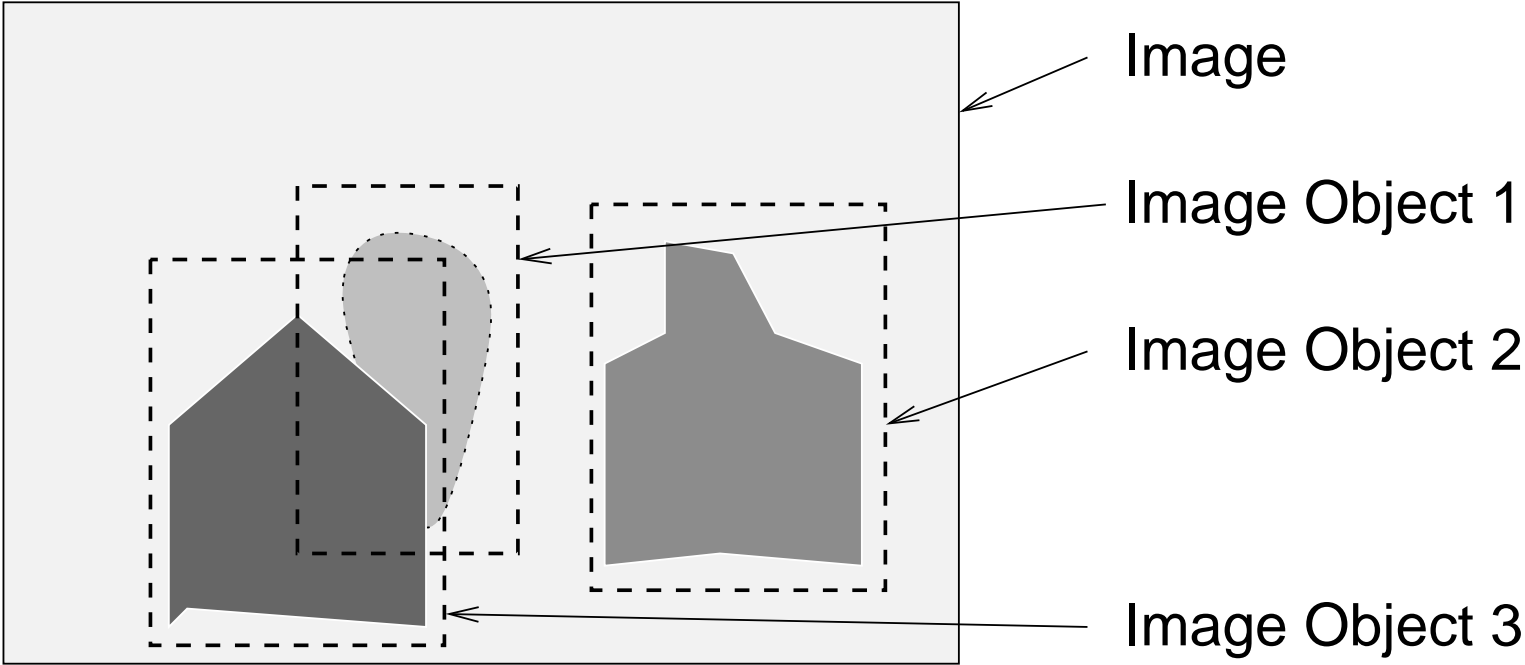
# X-Tree - number of Dimensions

---



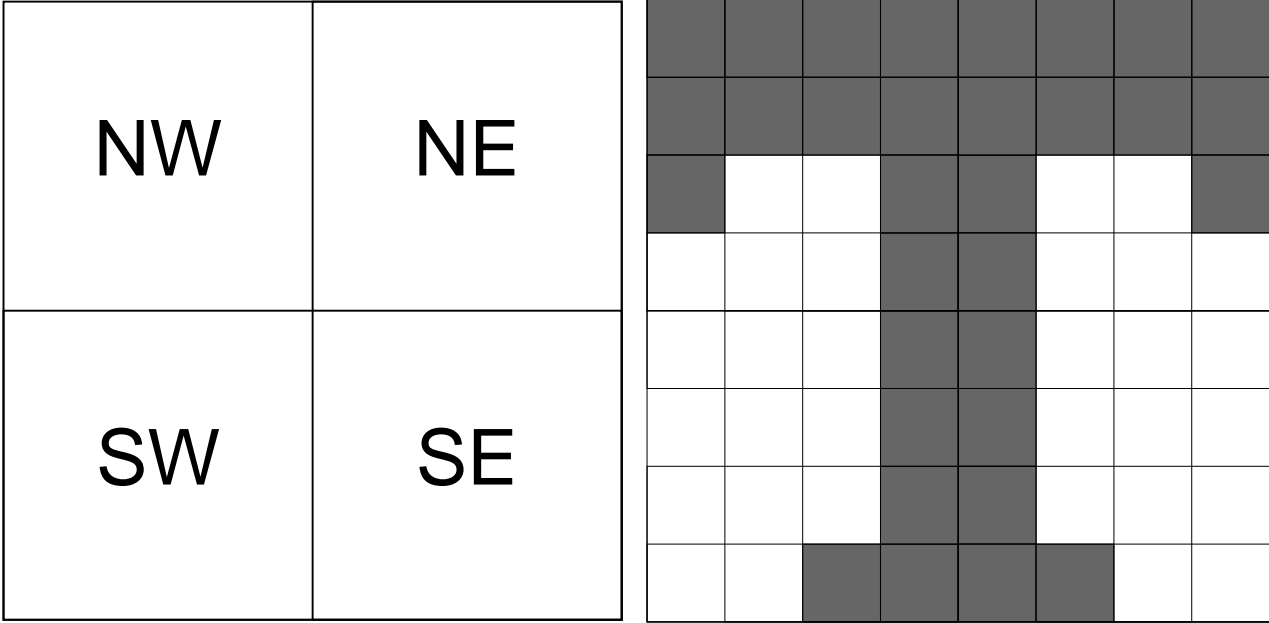
# Storage of Images

---



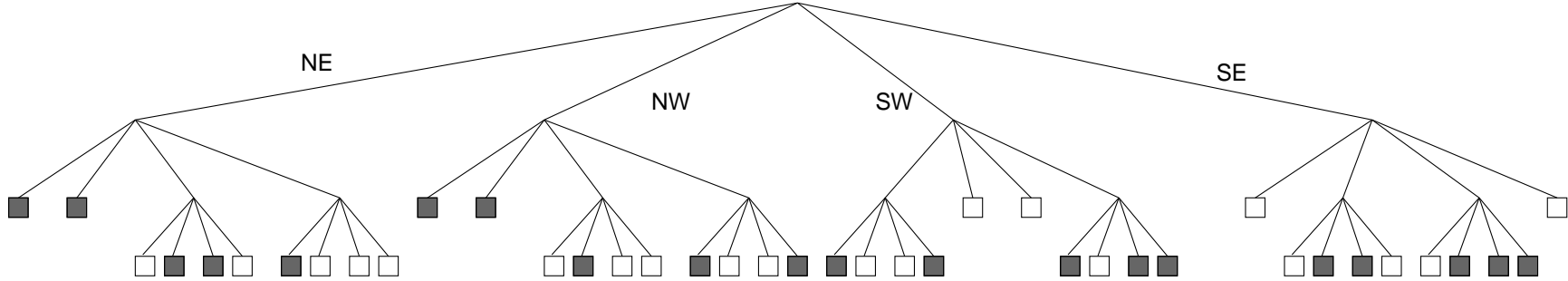
# Quadrees

---



# Quadtrees (II)

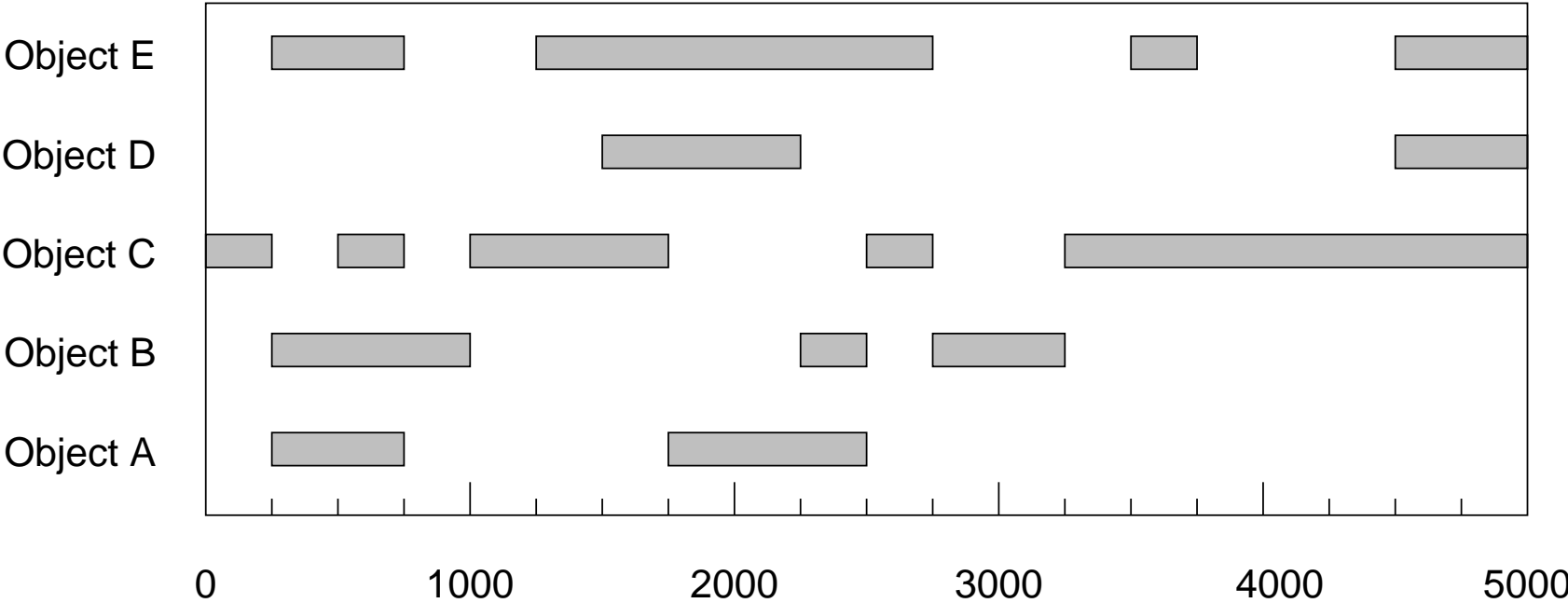
---



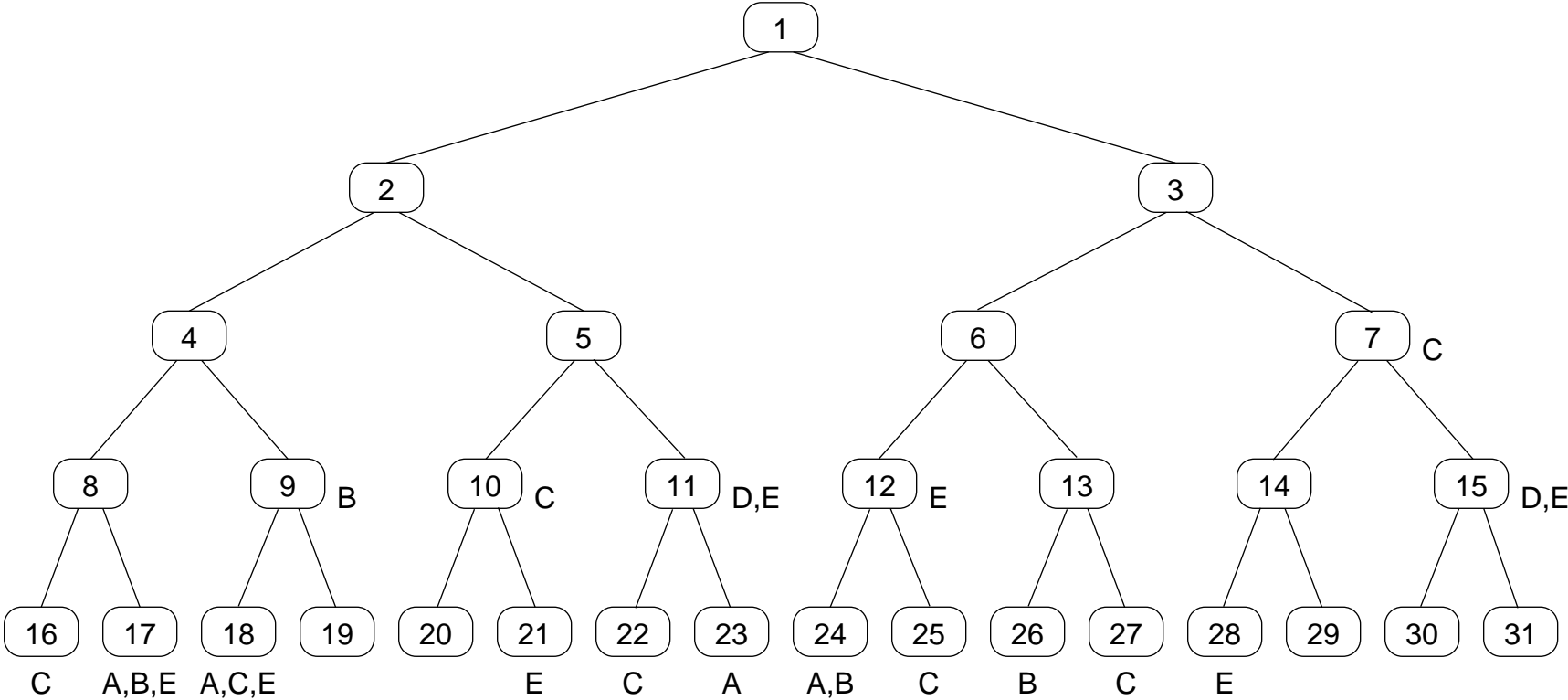
# Video Data

---

Video segments:



# Video Data: Segment-Tree



# Information Retrieval

---

- *Information Retrieval (IR):*

Technique for storage and *retrieval* of full text documents

- Examples:

```
[ 'Databases' ] in Text
```

```
[ 'Databases' and 'Multimedia' ] in Text
```

```
[ 'Object' 1 word prior to 'Orientation' ] in Text
```

```
[ 'Object' in same sentence as  
  'Orientation' ] in Text
```

```
[ 'Object' inside  
  2 sections with 'Orientation' ] in Text
```

# Quality Measures for IR

---

$$\text{Recall} = \frac{\text{number of located, relevant docs}}{\text{total number of relevant docs}}$$

$$\text{Precision} = \frac{\text{number of located, relevant docs}}{\text{total number of located docs}}$$

$$\text{Fallout} = \frac{\text{number of located, irrelevant docs}}{\text{total number of irrelevant docs}}$$

# Concept Indexing

---

- *Linguistic analysis*:
  - ◆ *morphologic analysis* generates a word stem by elimination of pre-/suffixes and plural endings → *Stemming*
  - ◆ Deriving new word forms from already known word forms → especially for german!
- *Synonyms*: binary relation between equivalent words
- *Term hierarchy* in (subject-specific) *thesaurus*:  
Inheritance of attributes → Queries to more general/specific terms possible

# Inverted Lists

---

- indexed words (strings) form a lexicographical sorted list
- Particular entry consists of a *word* and a list of document identifier which references to documents where the word occurs
- Additionally, further information for the word-document combination can be stored:
  - ◆ Position of (first occurrence of) word in text
  - ◆ Frequency of word in text

# Inverted Lists

---

