SIMILARITY SEARCH
The Metric Space Approach

Pavel Zezula, Giuseppe Amato,
Vlastislav Dohnal, Michal Batko

Similarity Search
The Metric Space Approach

Table of Contents

Features of "good" index structures

- Dynamicity
  - support insertions and deletions and minimize their costs
- Disk storage
  - for dealing with large collections of data
- CPU & I/O optimization
  - support different distance measures with completely different CPU requirements, e.g., L_2 and *quadratic-form distance.*
- Extensibility
  - similarity queries, i.e., range query, *k*-nearest neighbors query

Centralized Index Structures for Large Databases

1. **M-tree family**
2. hash-based metric indexing
3. performance trials

M-tree Family

- **The M-tree**
- Bulk-Loading Algorithm
- Multi-Way Insertion Algorithm
- The Slim Tree
- Slim-Down Algorithm
  - Generalized Slim-Down Algorithm
- Pivoting M-tree
- The M^+-tree
- The M^2-tree

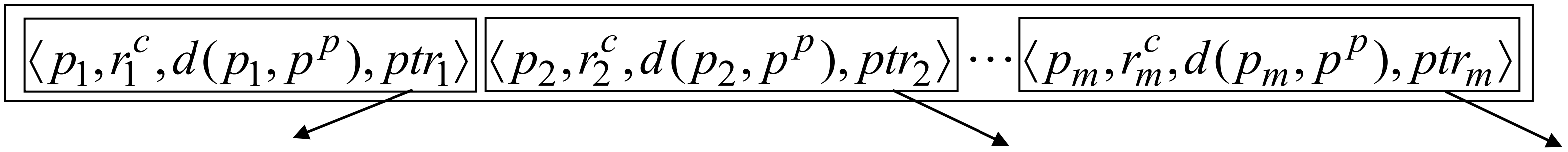
The M-tree

- Inherently dynamic structure
- Disk-oriented (fixed-size nodes)
- Built in a bottom-up fashion
  - Inspired by R-trees and B-trees

- All data in *leaf nodes*
- *Internal nodes*: pointers to subtrees and additional information
- Similar to GNAT, but objects are stored in leaves.

M-tree: Internal Node

- Internal node consists of an entry for each subtree
- Each entry consists of:
  - Pivot: *p*
  - Covering *radius* of the sub-tree: r^c
  - Distance from *p* to *parent* pivot p^p: $d(p,p^p)$
  - Pointer to sub-tree: *ptr*


$\langle p_1, r_1^c, d(p_1, p^p), ptr_1\rangle$
$\langle p_2, r_2^c, d(p_2, p^p), ptr_2\rangle$
$\cdots$
$\langle p_m, r_m^c, d(p_m, p^p), ptr_m\rangle$

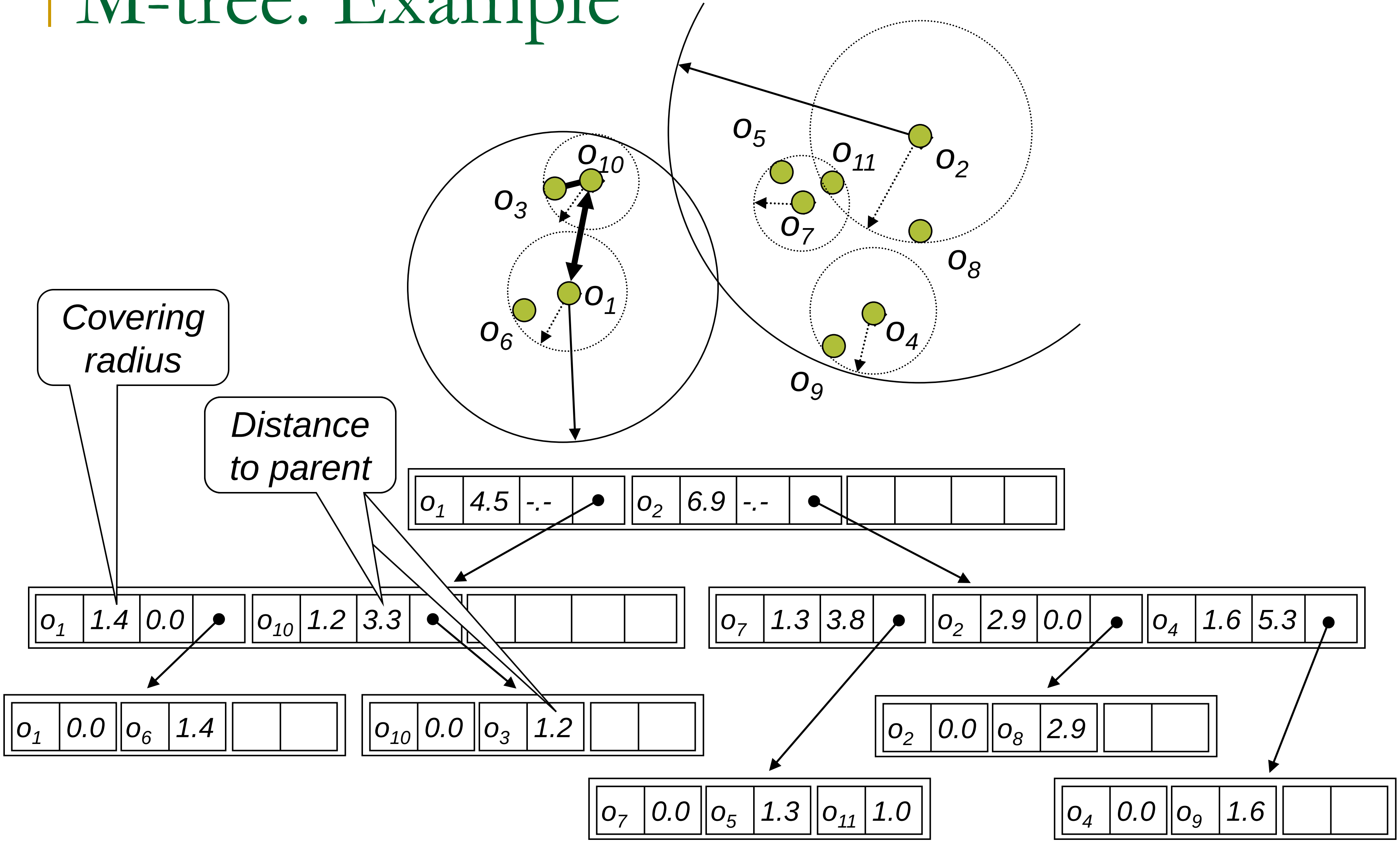
  - All objects in subtree *ptr* are within the distance r^c from *p.*

M-tree: Leaf Node

- leaf node contains **data entries**
- each entry consists of pairs:
  - object (its identifier): *o*
  - distance between *o* and its parent pivot: $d(o,o^p)$

$$\boxed{\boxed{\langle o_1, d(o_1,o^p)\rangle}\boxed{\langle o_2, d(o_2,o^p)\rangle}\cdots\boxed{\langle o_m, d(o_m,o^p)\rangle}}$$

M-tree: Example



Covering radius
Distance to parent
o1
o2
o3
o4
o5
o6
o7
o8
o9
o10
o11
o1 | 4.5 | -.- | o2 | 6.9 | -.-
o1 | 1.4 | 0.0 | o10 | 1.2 | 3.3
o7 | 1.3 | 3.8 | o2 | 2.9 | 0.0 | o4 | 1.6 | 5.3
o1 | 0.0 | o6 | 1.4
o10 | 0.0 | o3 | 1.2
o2 | 0.0 | o8 | 2.9
o7 | 0.0 | o5 | 1.3 | o11 | 1.0
o4 | 0.0 | o9 | 1.6

M-tree: Insert

- Insert a new object o_N:
- recursively descend the tree to locate the *most suitable leaf* for o_N
- in each step enter the subtree with pivot p for which:
  - no enlargement of radius r^c needed, i.e., $d(o_N,p) \le r^c$
    - in case of ties, choose one with p nearest to o_N
  - minimize the enlargement of r^c

M-tree: Insert (cont.)

- when reaching leaf node *N* then:
  - if *N* is not full then store o_N in *N*
  - else **Split**(N,o_N).

M-tree: Split

Split(N,o_N):

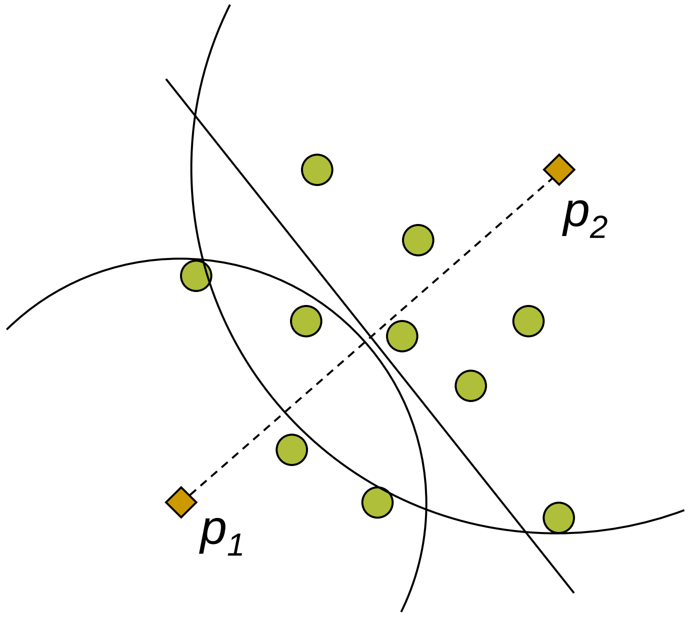
- Let S be the set containing all entries of N and o_N
- Select pivots p_1 and p_2 from S
- Partition S to S_1 and S_2 according to p_1 and p_2
- Store S_1 in N and S_2 in a new allocated node N'
- If N is root
  - Allocate a new root and store entries for p_1, p_2 there
- else (*let N^p and p^p be the parent node and parent pivot of N*)
  - Replace entry p^p with p_1
  - If N^p is full, then **Split**(N^p,p_2)
  - else store p_2 in node N^p

M-tree: Pivot Selection

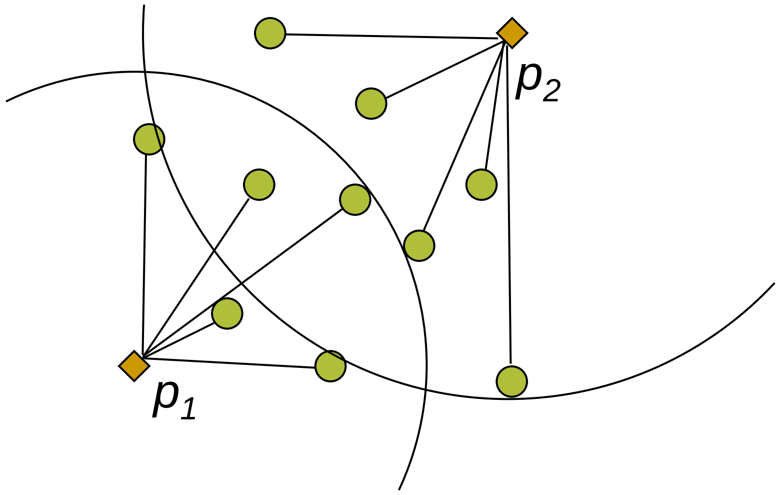
- Several pivots selection policies
  - ***RANDOM*** – select pivots p_1, p_2 randomly
  - ***m_RAD*** – select p_1, p_2 with minimum $(r_1^c + r_2^c)$
  - ***mM_RAD*** – select p_1, p_2 with minimum $max(r_1^c, r_2^c)$
  - ***M_LB_DIST*** – let $p_1 = p^p$ and $p_2 = o_i$ | $\max_\mathrm{i}$ { $d(o_i,p^p)$ }
    - Uses the pre-computed distances only
- Two versions (for most of the policies):
  - **Confirmed** – reuse the original pivot p^p and select only one
  - **Unconfirmed** – select two pivots (notation: ***RANDOM_2***)
- In the following, the ***mM_RAD_2*** policy is used.

M-tree: Split Policy

- Partition S to S_1 and S_2 according to p_1 and p_2
- Unbalanced
  - Generalized hyperplane



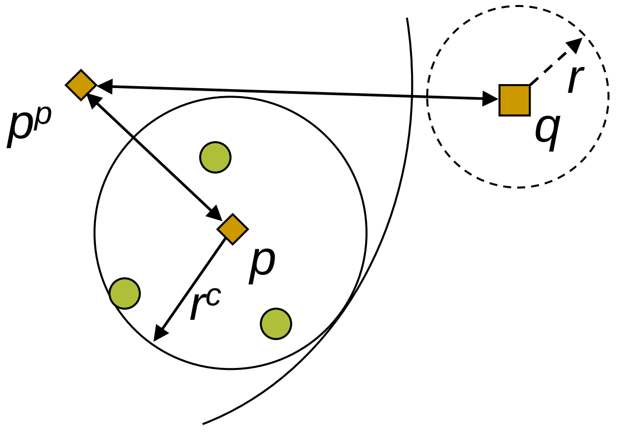
- Balanced
  - Larger covering radii
  - Worse than unbalanced one

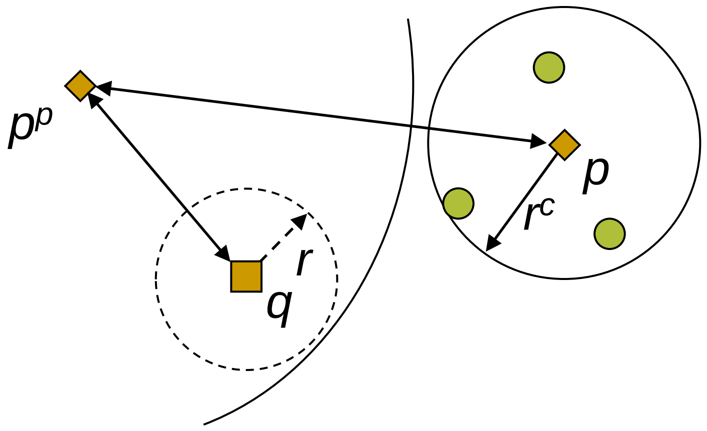


M-tree: Range Search

Given $R(q,r)$:

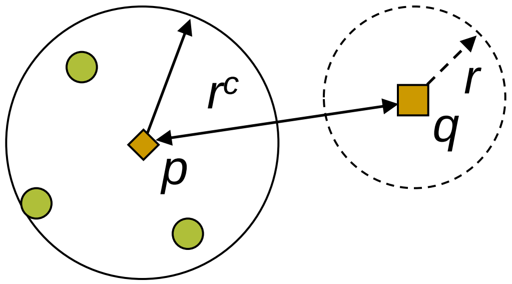
- Traverse the tree in a depth-first manner
- In an internal node, for each entry $\langle p,r^c,d(p,p^p),ptr\rangle$
  - Prune the subtree if $|d(q,p^p) - d(p,p^p)| - r^c > r$
  - Application of the pivot-pivot constraint





M-tree: Range Search (cont.)

- If not discarded, compute $d(q,p)$ and
  - Prune the subtree if $d(q,p) - r^c > r$
  - Application of the range-pivot constraint



- All non-pruned entries are searched recursively.

M-tree: Range Search in Leaf Nodes

- In a leaf node, for each entry $\langle o, d(o,o^p)\rangle$
  - Ignore entry if $|d(q,o^p) - d(o,o^p)| > r$
  - else compute $d(q,o)$ and check $d(q,o) \leq r$
  - Application of the object-pivot constraint

M-tree: *k*-NN Search

Given *k-NN*(*q*):

- Based on a *priority queue* and the pruning mechanisms applied in the range search.
- Priority queue:
  - Stores pointers to sub-trees where qualifying objects can be found.
  - Considering an entry $E=\langle p,r^c,d(p,p^p),ptr\rangle$, the pair $\langle ptr,d_{min}(E)\rangle$ is stored.
  - $d_{min}(E)=max\ \{\ d(p,q) - r^c,\ 0\ \}$
- Range pruning: instead of fixed radius *r*, use the distance to the *k*-th current nearest neighbor.

M-tree Family

- The M-tree
- **Bulk-Loading Algorithm**
- Multi-Way Insertion Algorithm
- The Slim Tree
- Slim-Down Algorithm
  - Generalized Slim-Down Algorithm
- Pivoting M-tree
- The M^+-tree
- The M^2-tree

Bulk-Loading Algorithm

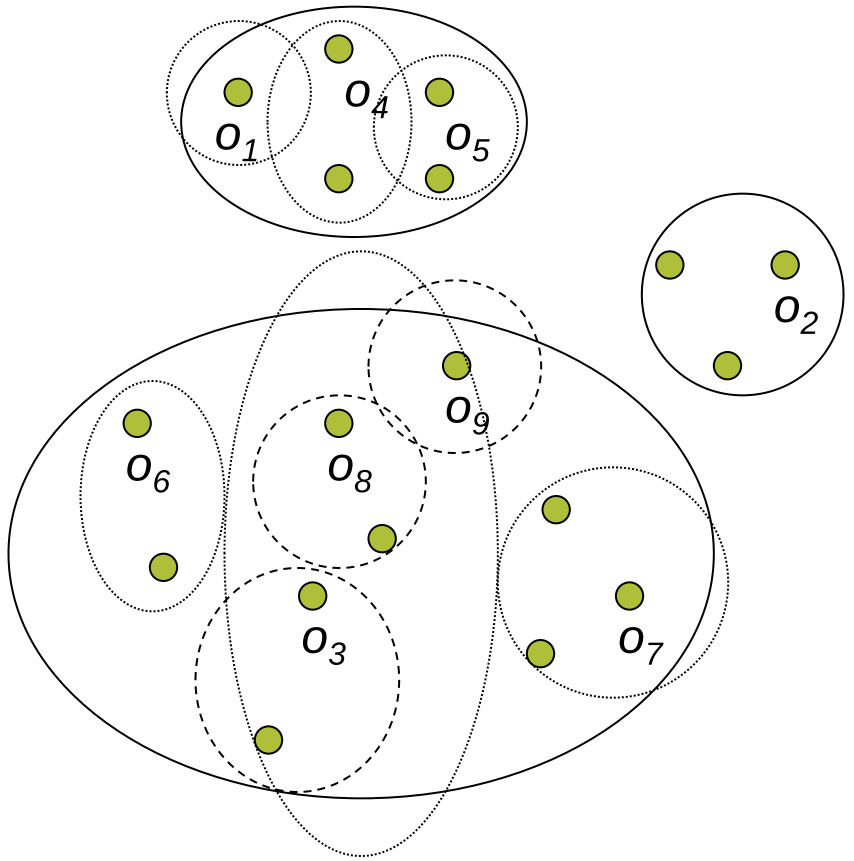
- first extension of M-tree
- improved tree-building (insert) algorithm
- requires the dataset to be *given in advance*

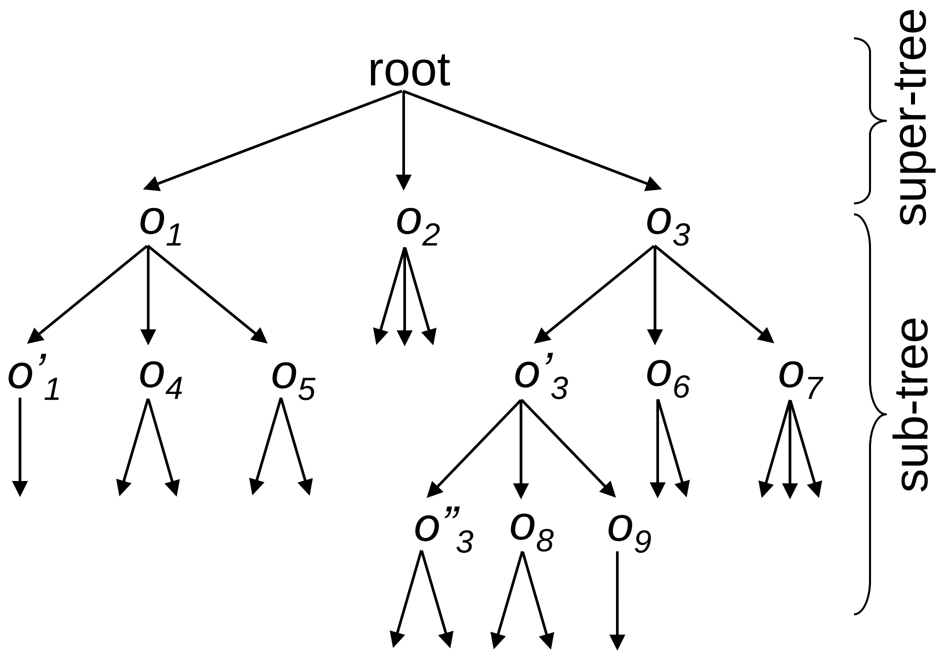
- **Notation:**
  - Dataset $X=\{o_1,\ldots,o_n\}$
  - Number of entries per node: m
- **Bulk-Loading Algorithm:**
  - First phase: build the M-tree
  - Second phase: refinement of unbalanced tree

Bulk-Loading: First Phase

- randomly select l pivots $P=\{p_1,\ldots,p_l\}$ from X
  - Usually $l=m$
- objects from X are assigned to the *nearest pivot* producing l subsets $P_1,\ldots,P_l$
- recursively apply the bulk-loading algorithm to the subsets and obtain l sub-trees $T_1,\ldots,T_l$
  - leaf nodes with maximally l objects
- create the *root* node and connect all the sub-trees to it.

Bulk-Loading: Example (1)





Bulk-Loading: Discussion

Problem of choosing pivots $P=\{p_1,\dots,p_l\}$

- sparse region $\rightarrow$ shallow sub-tree
  - far objects assigned to other pivots
- dense region $\rightarrow$ deep sub-tree

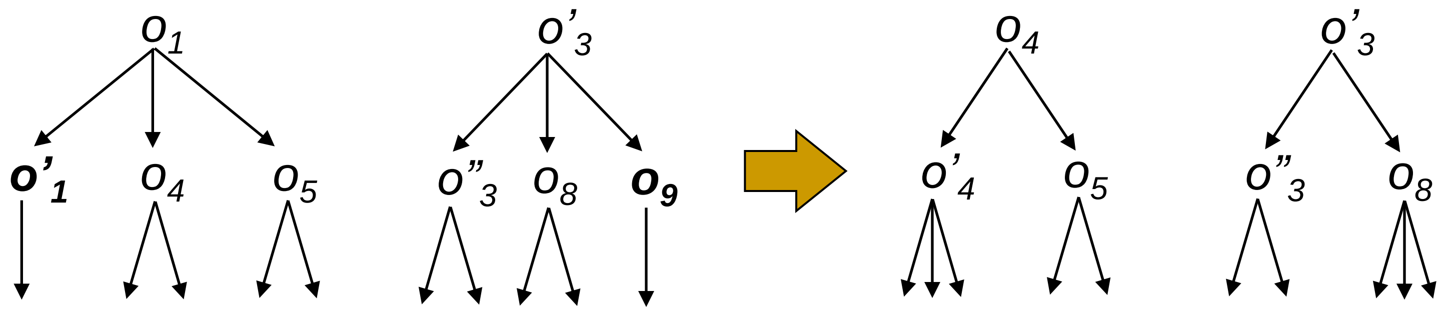
- observe this phenomenon in the example

Bulk-Loading: Second Phase

- refinement of the unbalanced M-tree
- apply the following two techniques to adjust the set of pivots $P=\{p_1,\ldots,p_l\}$
    - **under-filled nodes** – *reassign* to other pivots and *delete* corresponding pivots from *P*
    - **deeper subtrees** – *split* into shallower ones and *add* the obtained pivots to *P*

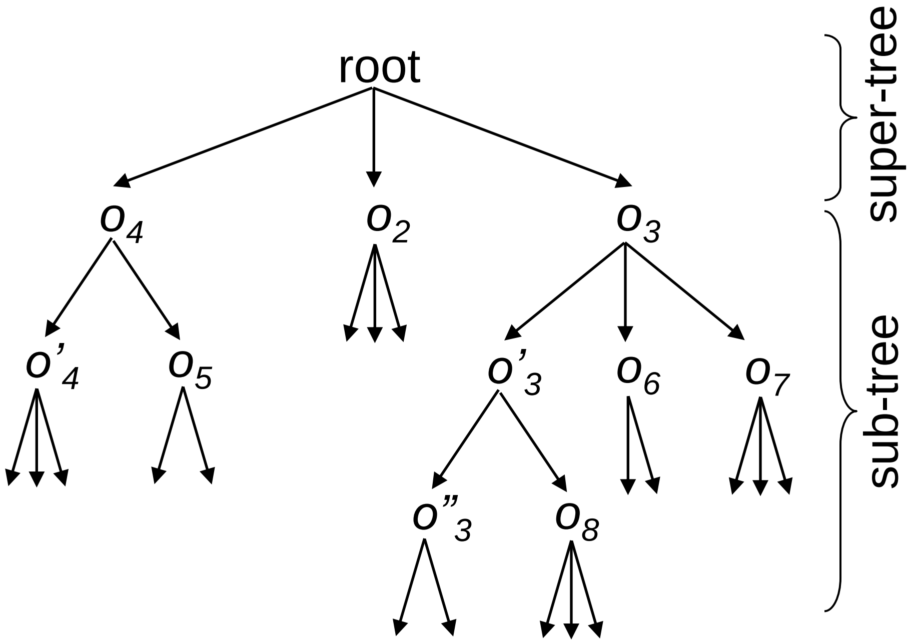
Bulk-Loading: Example (2)

- Under-filled nodes in the example: o'_1, o_9



Bulk-Loading: Example (3)

- After elimination of under-filled nodes.

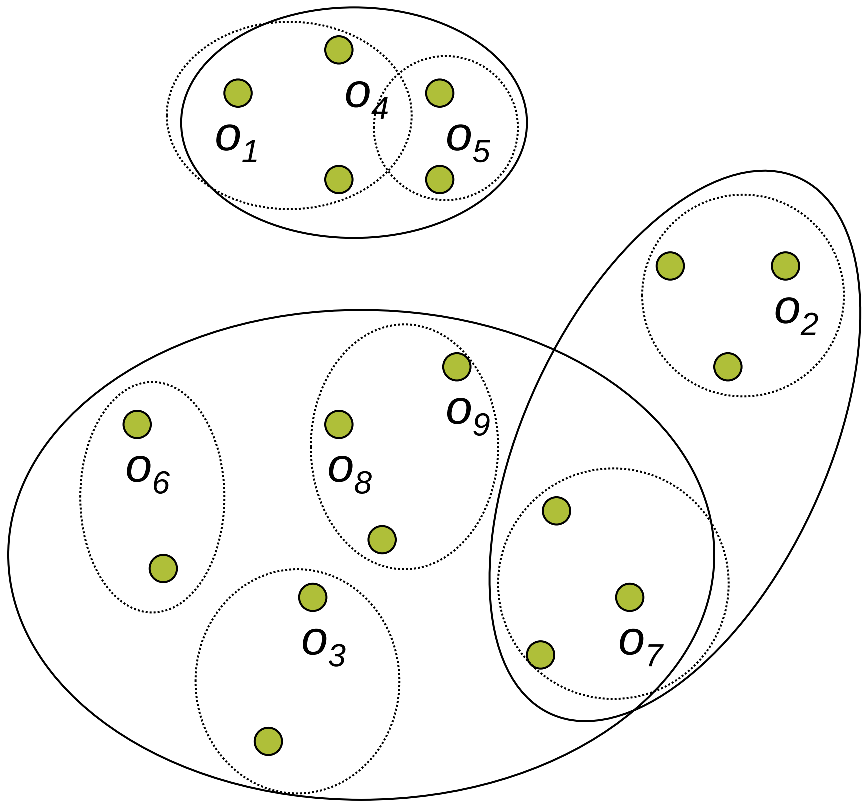


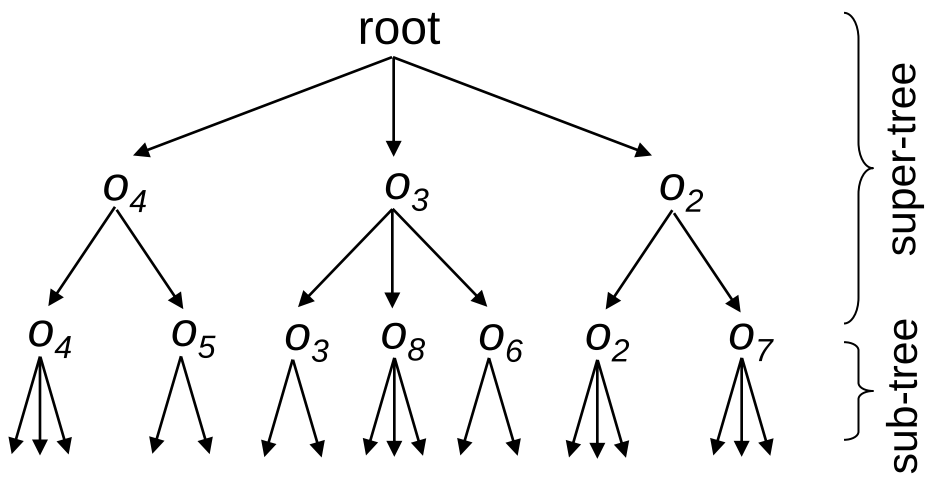
Bulk-Loading: Example (4)

- Sub-trees rooted in o_4 and o_3 in the tree are *deeper*

- split them into new subtrees rooted in o'_4, o_5, o''_3, o_8, o_6, o_7
- add them into P and remove o_4,o_3
- build the *super-tree* (two levels) over the final set of pivots $P=\{o_2,o'_4,o_5,o''_3,o_8,o_6,o_7\}$ – from Sample (3)

Bulk-Loading: Example (5) – Final





Bulk-Loading: Optimization

- Reduce the number of distance computations in the *recursive* calling of the algorithm
  - after initial phase, we have distances $d(p_j,o_i)$ for all objects $X=\{o_1,\ldots,o_n\}$ and all pivots $P=\{p_1,\ldots,p_l\}$
  - Assume the recursive processing of P_1
  - New set of pivots is picked $\{p_{1,1}, \ldots, p_{1,l'}\}$
  - During clustering, we are assigning every object $o \in P_1$ to its nearest pivot.
  - The distance $d(p_{1,j},o)$ can be lower-bounded:

$$|d(p_1,o) - d(p_1,p_{1,j})| \leq d(p_{1,j},o)$$

Bulk-Loading: Optimization (cont.)

- If this lower-bound is greater than the distance to the closest pivot $p_{1,N}$ so far, i.e.,

  $$|d(p_1,o) - d(p_1,p_{1,j})| > d(p_{1,N},o)$$

  then the evaluation of $d(p_{1,j},o)$ can be avoided.

- Cuts costs by 11%
  - It uses pre-computed distances to a single pivot.
  - by 20% when pre-computed distances to multiple pivots are used.

M-tree Family

- The M-tree
- Bulk-Loading Algorithm
- **Multi-Way Insertion Algorithm**
- The Slim Tree
- Slim-Down Algorithm
  - Generalized Slim-Down Algorithm
- Pivoting M-tree
- The M^+-tree
- The M^2-tree

Multi-Way Insertion Algorithm

- another extension of M-tree insertion algorithm
- objective: *build more compact trees*
  - reduce search costs (both I/O and CPU)
- for dynamic datasets (not necessarily given in advance)
- increase insertion costs slightly
- the original *single-way* insertion visits exactly one root-leaf branch
  - leaf with *no* or *minimum* increase of covering radius
  - not necessarily the most convenient

Multi-Way Insertion: Principle

- when inserting an object o_N
- run the *point query* $R(o_N,0)$
- for all visited leaves (they can store o_N without radii enlargement): compute the distance between o_N and the leaf's pivot
- choose the closest pivot (leaf)
- if no leaf visited – run the single-way insertion

Multi-Way Insertion: Analysis

Insertion costs:

- 25% higher I/O costs (more nodes examined)
- higher CPU costs (more distances computed)

Search costs:

- 15% fewer disk accesses
- almost the same CPU costs for the *range query*
- 10% fewer distance computations for *k-NN query*

M-tree Family

- The M-tree
- Bulk-Loading Algorithm
- Multi-Way Insertion Algorithm
- **The Slim Tree**
- Slim-Down Algorithm
  - Generalized Slim-Down Algorithm
- Pivoting M-tree
- The M$^+$-tree
- The M^2-tree

The Slim Tree

- extension of M-tree – the same structure
  - speed up insertion and node splitting
  - improve storage utilization
- new **node-selection heuristic** for insertion
- new **node-splitting** algorithm
- special **post-processing** procedure
  - make the resulting trees more compact.

Slim Tree: Insertion

Starting at the root node, in each step:

- find a node that *covers* the incoming object
- if none, select the node whose pivot is *the nearest*
    - M-tree would select the node whose covering radius requires the smallest expansion
- if several nodes qualify, select the one which occupies the minimum space
    - M-trees would choose the node with closest pivot

Slim Tree: Insertion Analysis

- fill insufficiently occupied nodes first
  - defer splitting, boost node utilization, and cut the tree size
- experimental results (the same ***mM_RAD_2*** splitting policy) show:
  - lower I/O costs
  - nearly the same number of distance computations
  - this holds for both the *tree building procedure* and the *query execution*

Slim Tree: Node Split

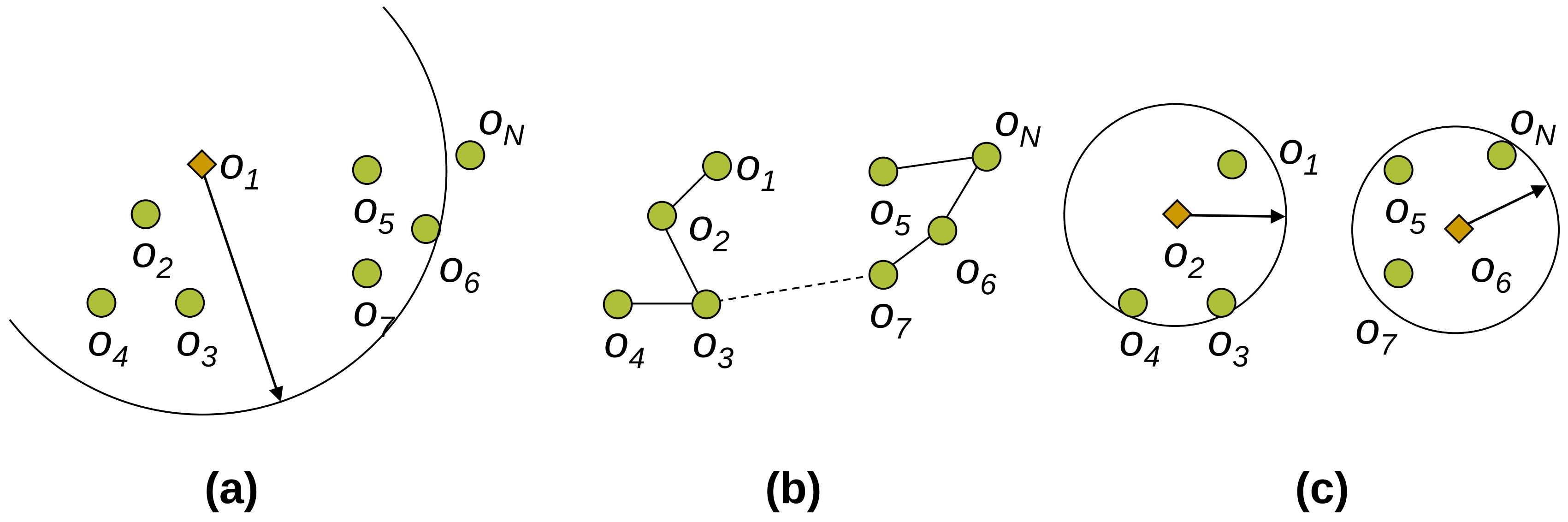
- splitting of the overfilled nodes – high costs
- ***mM_RAD_2*** strategy is considered the best so far
    - Complexity $O(n^3)$ using $O(n^2)$ distance computations
- the Slim Tree splitting based on the *minimum spanning tree* (MST)
    - Complexity $O(n^2 \log n)$ using $O(n^2)$ distance computations
- the MST algorithm assumes a full graph
    - n objects
    - $n(n-1)$ edges – distances between objects

Slim Tree: Node Split (cont.)

Splitting policy based on the MST:

1. build the *minimum spanning tree* on the full graph
2. delete the *longest edge*
3. the two resulting sub-graphs form the *new nodes*
4. choose the *pivot* for each node as the *object* whose distance to the *others* in the group is *the shortest*

Slim Tree: Node Split – Example



- (a) the original Slim Tree node
- (b) the minimum spanning tree
- (c) the new two nodes

Slim Tree: Node Split – Discussion

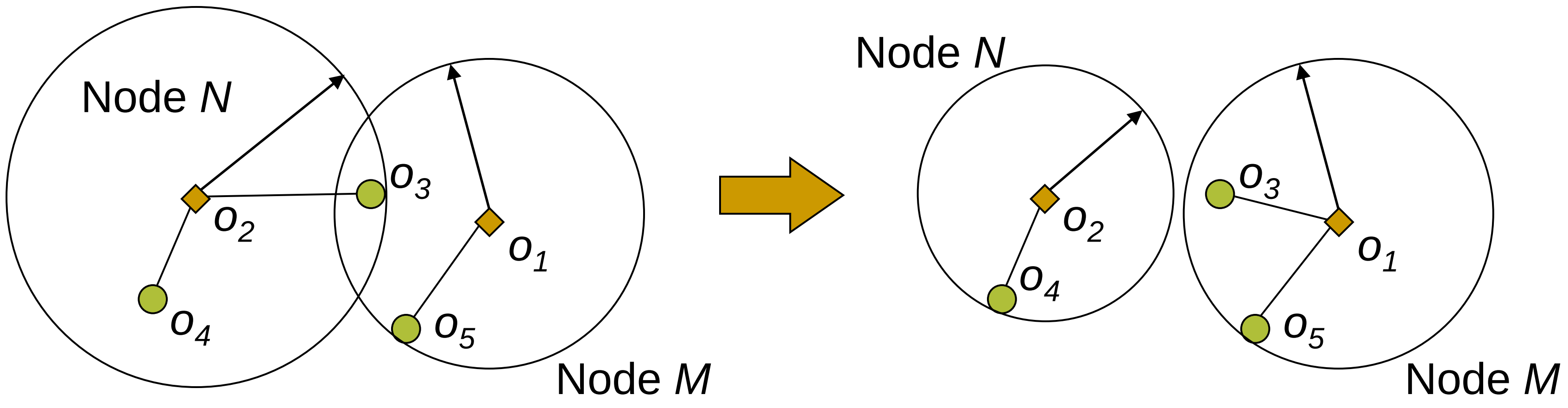
- does not guarantee the balanced split
- a possible variant (more balanced splits):
  - choose the most appropriate edge from among *the longer edges* in the MST
  - if no such edge is found (e.g., for a star-shaped dataset), accept the original unbalanced split
- experiments prove that:
  - tree building using the MST algorithm is at least forty times faster than the ***mM_RAD_2*** policy
  - query execution time is not significantly better

M-tree Family

- The M-tree
- Bulk-Loading Algorithm
- Multi-Way Insertion Algorithm
- The Slim Tree
- **Slim-Down Algorithm**
  - Generalized Slim-Down Algorithm
- Pivoting M-tree
- The M$^+$-tree
- The M^2-tree

Slim-Down Algorithm

- post-processing procedure
- reduce the *fat-factor* of the tree
  - basic idea: reduce the overlap between nodes on one level
  - minimize number of nodes visited by a point query, e.g., $R(o_3,0)$



Slim-Down Algorithm: The Principle

For each node *N* at the leaf level:

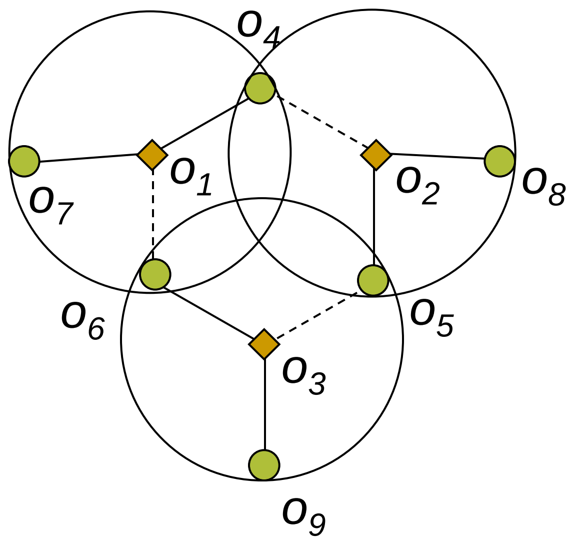
1. Find object *o* furthest from pivot of *N*
2. Search for a sibling node *M* that also covers *o.*
   If such a not-fully-occupied node exists, move *o* from *N* to *M* and update the covering radius of *N.*

- Steps 1 and 2 are applied to all nodes at the given level. If an object is relocated after a complete loop, the entire algorithm is executed again.

- Observe moving of o_3 from *N* to *M* on previous slide.

Slim-Down Algorithm: Discussion

- Prevent from infinite loop
  - cyclic moving of objects o_4, o_5, o_6
- Limit the number of algorithm cycles



- Trials proved reducing of I/O costs of at least 10%
- The idea of *dynamic object relocation* can be also applied to *defer splitting.*
  - Move distant objects from a node instead of splitting it.

M-tree Family

- The M-tree
- Bulk-Loading Algorithm
- Multi-Way Insertion Algorithm
- The Slim Tree
- **Slim-Down Algorithm**
  - **Generalized Slim-Down Algorithm**
- Pivoting M-tree
- The M^+-tree
- The M^2-tree

Generalized Slim-Down Algorithm

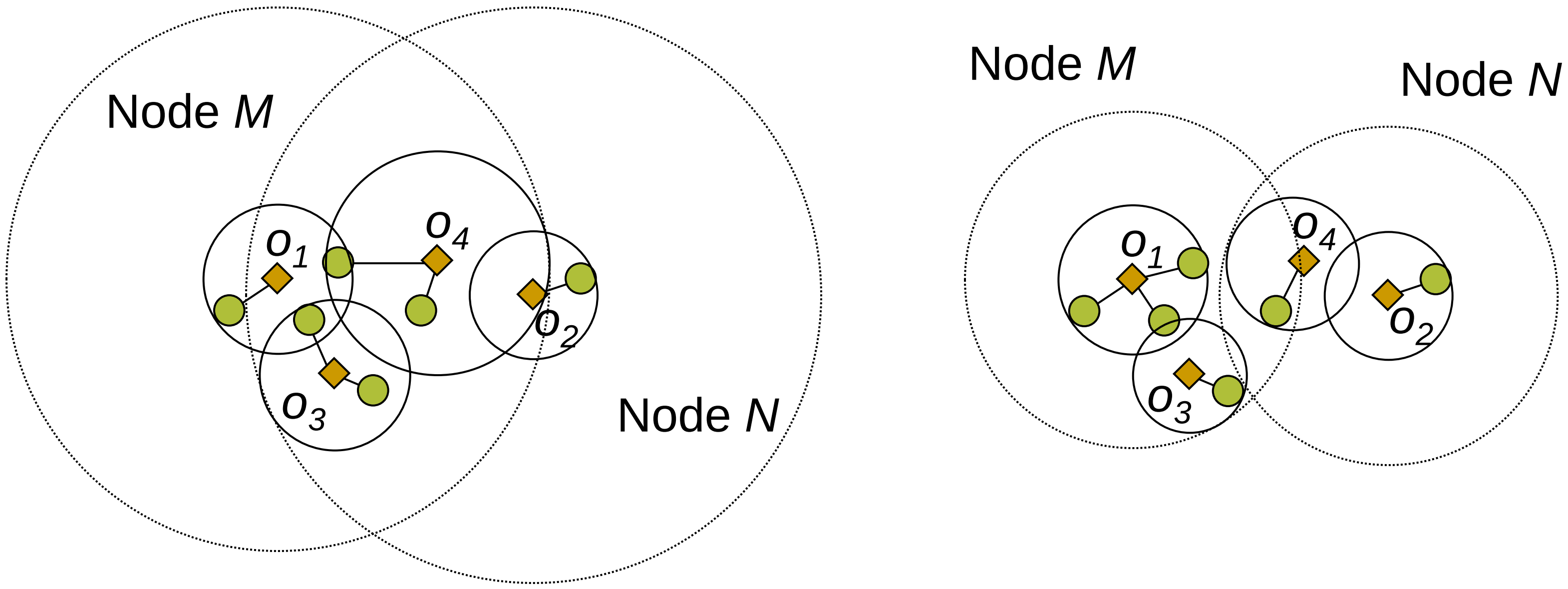
- generalization of Slim-down algorithm for non-leaf tree levels
- the covering radii r^c must be taken into account before moving a non-leaf entry
- the generalized Slim-down starts from the leaf level
  - follow the original Slim-down algorithm for leaves
- ascend up the tree terminating in the root

Generalized Slim-Down: The Principle

For each entry $E=\langle p,r^c,\ldots\rangle$ at given non-leaf level:

- pose range query $R(p,r^c)$,
- the query determines the set of nodes that *entirely contain* the query region,
- from this set, choose the node M whose parent pivot is closer to p than to p^p,
- if such M exists, move the entry E from N to M,
- if possible, shrink the covering radius of N.

Generalized Slim-Down: Example



- Leaf level:
  - move two objects from o_3 and o_4 to o_1 – shrink o_3 and o_4
- Upper level:
  - originally node M contains o_1,o_4 and node N contains o_2,o_3
  - swap the nodes of o_3 and o_4

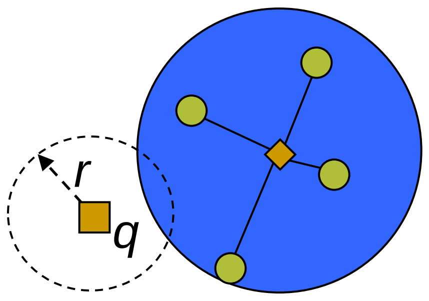
M-tree Family

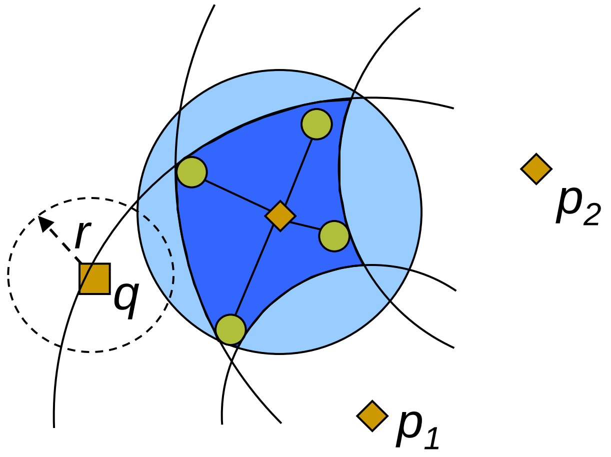
- The M-tree
- Bulk-Loading Algorithm
- Multi-Way Insertion Algorithm
- The Slim Tree
- Slim-Down Algorithm
    - Generalized Slim-Down Algorithm
- **Pivoting M-tree**
- The M$^+$-tree
- The M^2-tree

Pivoting M-tree

- upgrade of the standard M-tree
- bound the region covered by nodes more tightly
  - define additional *ring regions* that restrict the ball regions
  - ring regions: pivot *p* and two radii r_{min}, r_{max}
  - such objects *o* that: $r_{min} \leq d(o,p) \leq r_{max}$
- basic idea:
  - Select additional *pivots*
  - Every pivot defines two boundary values between which all node’s objects lie.
  - Boundary values for each pivot are stored in every node. (see a motivation example on the next slide)

PM-tree: Motivation Example





- original M-tree
- range query $R(q,r)$ intersects the node region

- PM-tree (two pivots)
- this node not visited for query $R(q,r)$

PM-tree: Structure

- select additional set of pivots $|P|=n_p$
- leaf node entry: $\langle o,d(o,o^p),PD\rangle$
  - PD – array of n_{pd} pivot distances: $PD[i]=d(p_i,o)$
  - Parameter $n_{pd} < n_p$
- internal node entry: $\langle p,r^c,d(p,p^p),ptr,HR\rangle$
  - HR – array of n_{hr} intervals defining ring regions

$$HR[j].\min = \min(\{d(o,p_j) \mid \forall o \in ptr\})$$

$$HR[j].\max = \max(\{d(o,p_j) \mid \forall o \in ptr\})$$

  - parameter $n_{hr} < n_p$

PM-tree: Insertion

- insertion of object o_N
- the *HR* arrays of nodes visited during insertion must be updated by values $d(o_N,p_i)$ for all $i \leq n_{hr}$
- the leaf node:
  - create array *PD* and fill it with values $d(o_N,p_j)$, $\forall\ j \leq n_{pd}$
- values $d(o_N,p_j)$ are computed only *once* and used *several* times – $max(n_{hr}, n_{pd})$ distance computations
- insertions may force *node splits*

PM-tree: Node Split

- node splits require some maintenance
- leaf split:
  - set arrays *HR* of two new internal entries
  - set *HR*[*i*].*min* and *HR*[*i*].*max* as min/max of *PD*[*j*]
  - compute additional distances: $d(p_j, o)$, $\forall j \ (n_{pd} < j \leq n_{hr})$ and take them into account
  - can be expensive if $n_{hr} >> n_{pd}$
- internal node split:
  - creating two internal node entries with *HR*
  - set these *HR* arrays as *union* over all *HR* arrays of respective entries

PM-tree: Range Query

Given $R(q,r)$:

- evaluate distances $d(q,p_i)$, $\forall\ i\ (i \leq max(n_{hr}, n_{pd}))$
- traverse the tree, internal node $\langle p, r^c, d(p,p^p), ptr, HR \rangle$ is visited if both the expressions hold:

$$d(q,p) \leq r + r^c$$

$$\bigwedge_{i=1}^{n_{hr}} (d(q,p_i) - r \leq HR[i].\max \wedge d(q,p_i) + r \geq HR[i].\min)$$

- leaf node entry test: $\bigwedge_{i=1}^{n_{pd}} (|d(q,p_i) - PD[i]| \leq r)$
- M-tree: the first condition only

PM-tree: Parameter Setting

- general statements:
  - existence of *PD* arrays in leaves reduce number of distance computations but increase the I/O cost
  - the *HR* arrays reduce both CPU and I/O costs
- experiments proof that:
  - $n_{pd}=0$ decreases I/O costs by 15% to 35% comparing to M-tree (for various values of n_{hr})
  - CPU cost reduced by about 30%
  - $n_{pd}=n_{hr}/4$ leads to the same I/O costs as for M-tree
  - with this setting – up to 10 times faster
- particular parameter setting depends on application

M-tree Family

- The M-tree
- Bulk-Loading Algorithm
- Multi-Way Insertion Algorithm
- The Slim Tree
- Slim-Down Algorithm
    - Generalized Slim-Down Algorithm
- Pivoting M-tree
- **The M$^+$-tree**
- The M^2-tree

The M$^+$-tree

- modification of the M-tree
- restrict the application to L_p metrics (vector spaces)
- based on the concept of *key dimension*
- each node partitioned into two *twin-nodes*
  - partition according to a selected *key dimension*

M$^+$-tree: Principles

- in an *n*-dimensional vector space
- *key dimension* for a set of objects is the dimension along which the data objects are *most spread*
- for any dimension D_{key} and vectors $(x_1,\ldots x_n),(y_1,\ldots y_n)$

$$|x_{D_{key}} - y_{D_{key}}| \le \sqrt{(x_1-y_1)^2+\cdots+(x_n-y_n)^2}$$

- this holds also for other L_p metrics
- this fact is applied to prune the search space

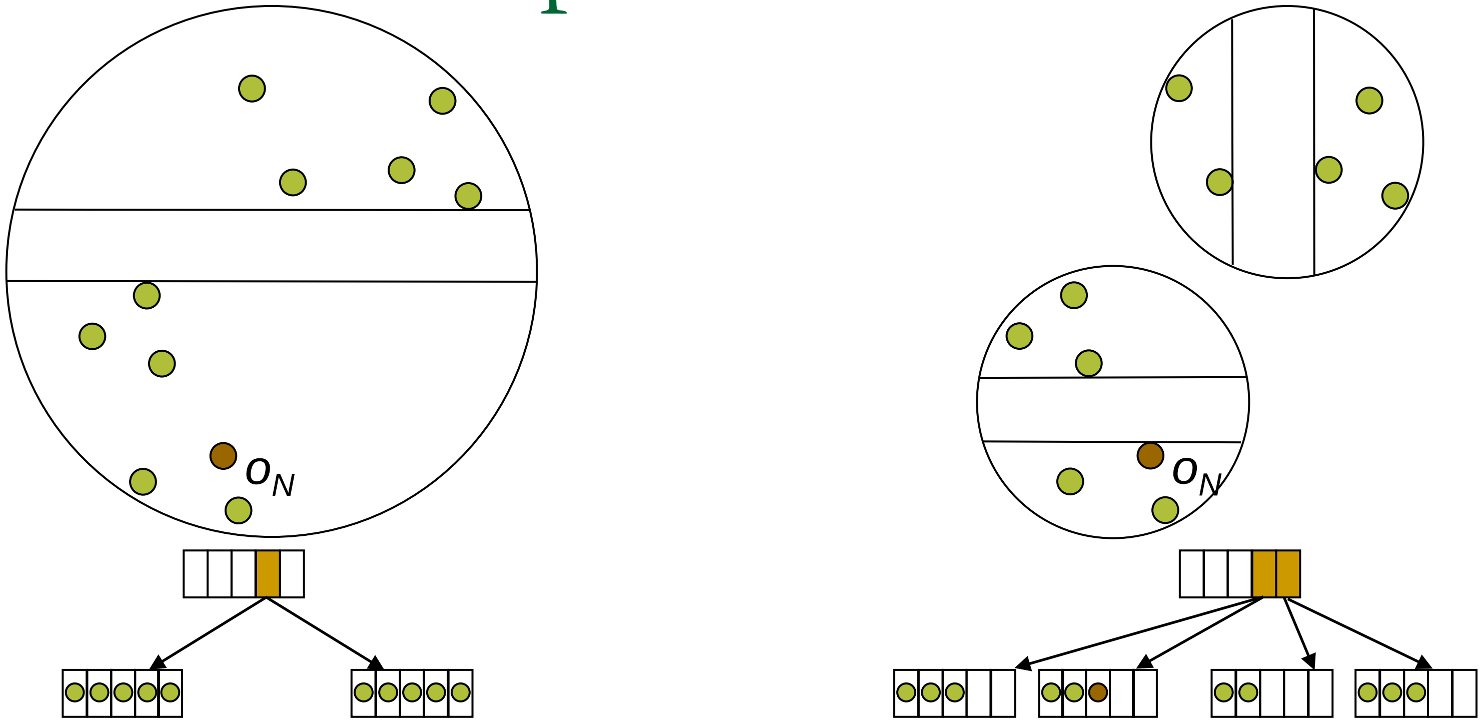
M^{+}-tree: Structure

- internal node is divided into two subsets
  - according to a selected dimension
  - leaving a *gap* between the two subsets
  - the greater the gap the better filtering
- internal node entry:

  $$\langle p, r^{c}, d(p, p^{p}), D_{key}, ptr_{left}, d_{lmax}, d_{rmin}, ptr_{right} \rangle$$

  - D_{key} – number of the key dimension
  - ptr_{left} ,ptr_{right} – pointers to the left and right twin-nodes
  - d_{lmax} – *maximal* key-dimension value of the left twin
  - d_{rmin} – *minimal* key-dimension value of the right twin

M$^+$-tree: Example



- splitting of an overfilled node:
  - objects of both twins are considered as a single set
  - apply standard ***mM_RAD_2*** strategy
- select the *key dimension* for each node separately

M$^+$-tree: Performance

- slightly more efficient than M-tree
- better filtering for range queries with small radii
- practically the same for larger radii
- nearest neighbor queries:
  - a shorter *priority queue* – only one of the twin-nodes
  - save some time for queue maintenance

- moderate performance improvements
- application restricted to vector datasets with L_p

M-tree Family

- The M-tree
- Bulk-Loading Algorithm
- Multi-Way Insertion Algorithm
- The Slim Tree
- Slim-Down Algorithm
  - Generalized Slim-Down Algorithm
- Pivoting M-tree
- The M$^+$-tree
- **The M^2-tree**

The M^2-tree

- generalization of M-tree
- able to process *complex similarity queries*
  - combined queries on several metrics at the same time
  - for instance: an image database with keyword-annotated objects and color histograms
  - query: *Find images that contain a lion and the scenery around it like this.*
- qualifying objects identified by a *scoring function* d_f
  - combines the particular distances (according to several different measures)

M^2-tree: Structure

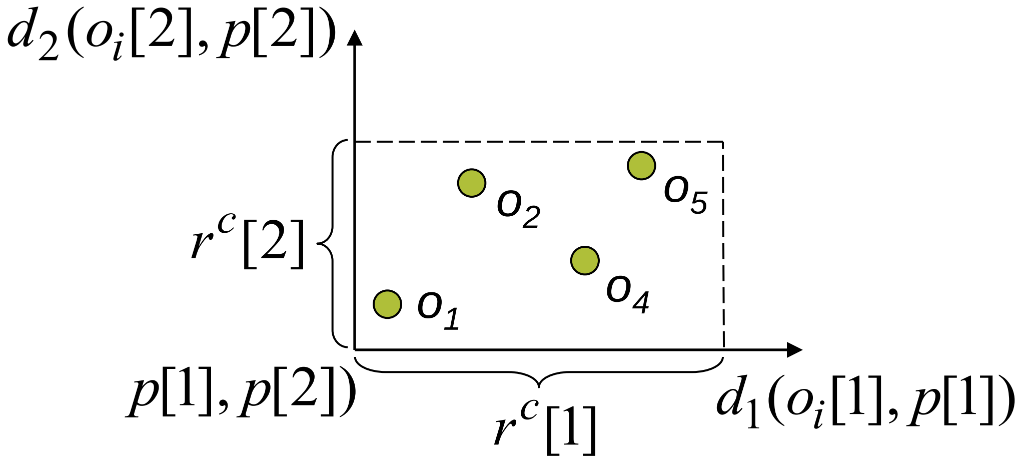
- each object characterized by several *features*
  - e.g. *o[1],o[2]*
  - respective distance measures may differ: d_1,d_2
- leaf node: M-tree vs. M^2-tree

  $\langle o, d(o,p)\rangle$ $\langle o[1], d_1(o[1], p[1]), o[2], d_2(o[1], p[2])\rangle$
- internal node: M-tree vs. M^2-tree

$$\langle p, r^c, d(p, p^p), ptr\rangle$$

$$\langle p[1], r^c[1], d_1(p[1], p^p[1]), p[2], r^c[2], d_2(p[2], p^p[2]), ptr\rangle$$

M^2-tree: Example



- the space transformation according to particular features can be seen as an *n*-dimensional space
- the subtree region forms a *hypercube*

M^2-tree: Range Search

Given R(q,r):

- M-tree prunes a subtree if $|d(q,p^p) - d(p,p^p)| - r^c > r$
- M^2-tree: compute the lower bound for every feature

  $$\forall i, \min(|\, d_i(q[i], p^p[i]) - d_i(p[i], p^p[i])\,| - r^c[i], 0)$$

- combine these bounds using the scoring function d_f
- visit those entries for which the result is $\leq r$

- analogous strategy for nearest neighbor queries

M^2-tree: Performance

- running *k-NN* queries
- image database mentioned in the example
- M^2-tree compared with *sequential scan*
  - the same I/O costs
  - reduced number of distance computations
- M^2-tree compared with Fagin's $\mathcal{A}_0$ (two M-trees)
  - M^2-tree saves about 30% of I/Os
  - about 20% of distance computations
  - $\mathcal{A}_0$ have higher I/O cost than the sequential scan

Centralized Index Structures for Large Databases

1. M-tree family

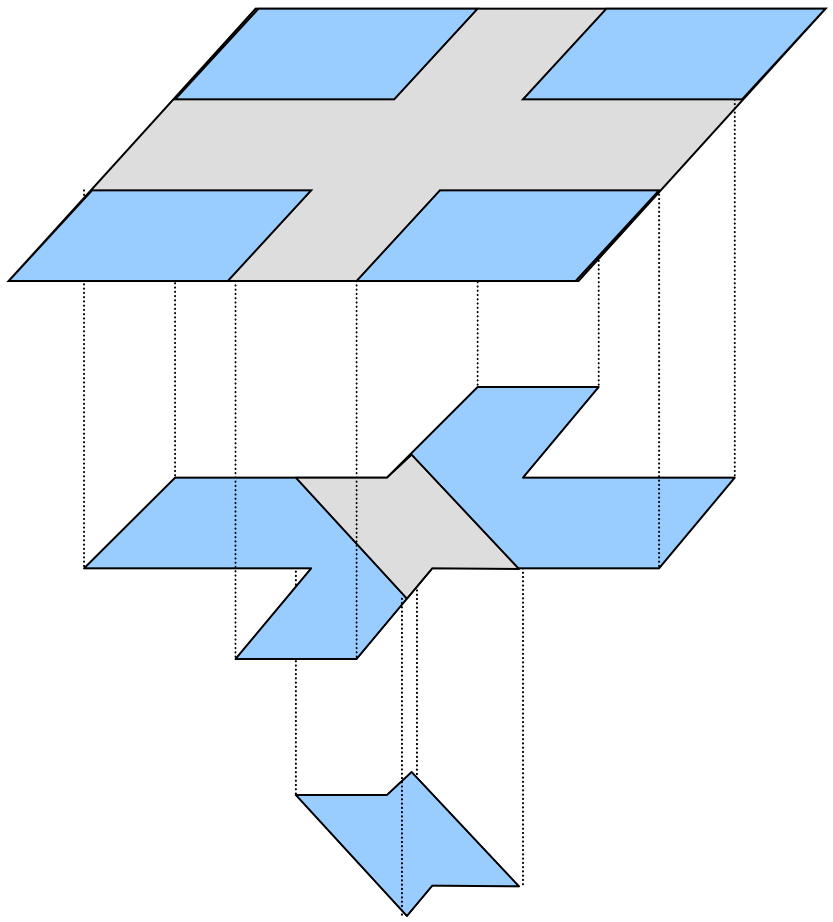
2. **hash-based metric indexing**
   - Distance Index (D-index)
   - Extended D-Index (eD-index)

3. performance trials

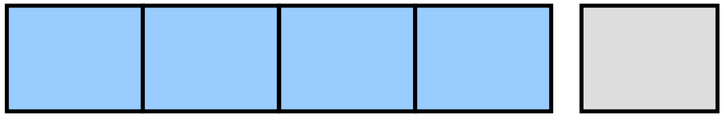
Distance Index (D-index)

- Hybrid structure
  - combines pivot-filtering and partitioning.
- Multilevel structure based on hashing
  - one ρ-split function per level.
- The first level splits the whole data set.
- Next level partitions the exclusion zone of the previous level.
- The exclusion zone of the last level forms the exclusion bucket of the whole structure.

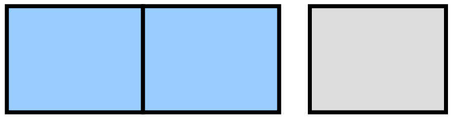
D-index: Structure



4 separable buckets at the first level



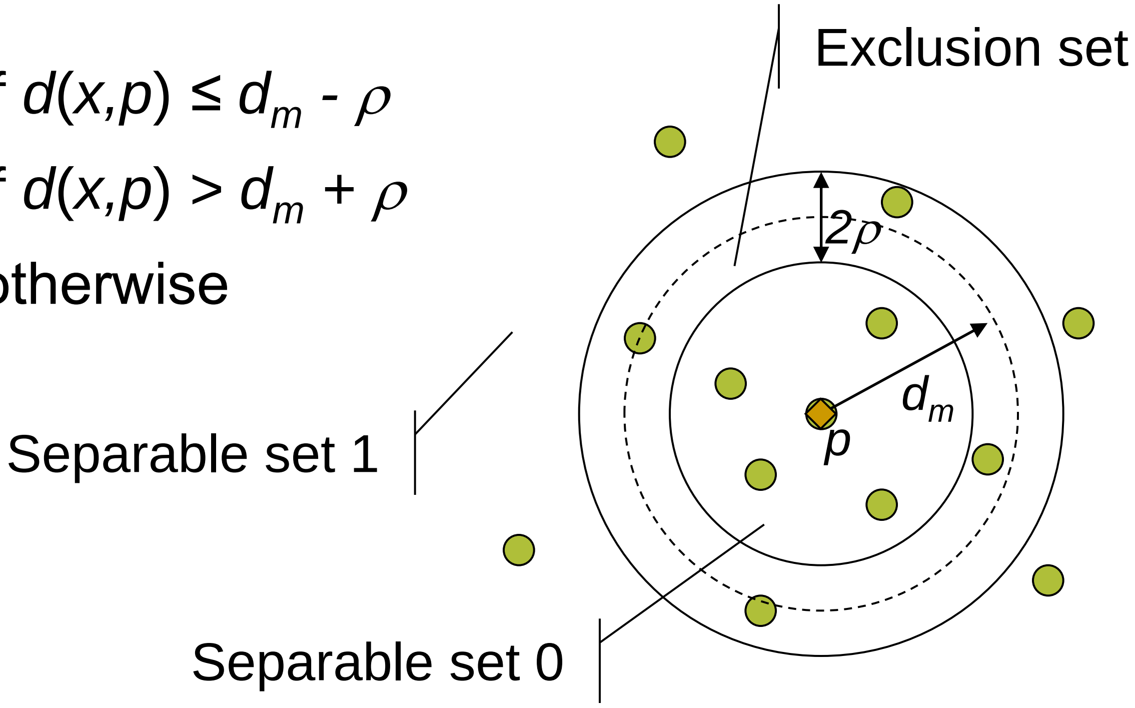
2 separable buckets at the second level



exclusion bucket of the whole structure

D-index: Partitioning

- Based on excluded middle partitioning
  - ball partitioning variant is used.
  - $bps^{1,\rho}(x) = \begin{cases} 0 & \text{if } d(x,p) \leq d_m - \rho \\ 1 & \text{if } d(x,p) > d_m + \rho \\ - & \text{otherwise} \end{cases}$



D-index: Binary ρ-Split Function

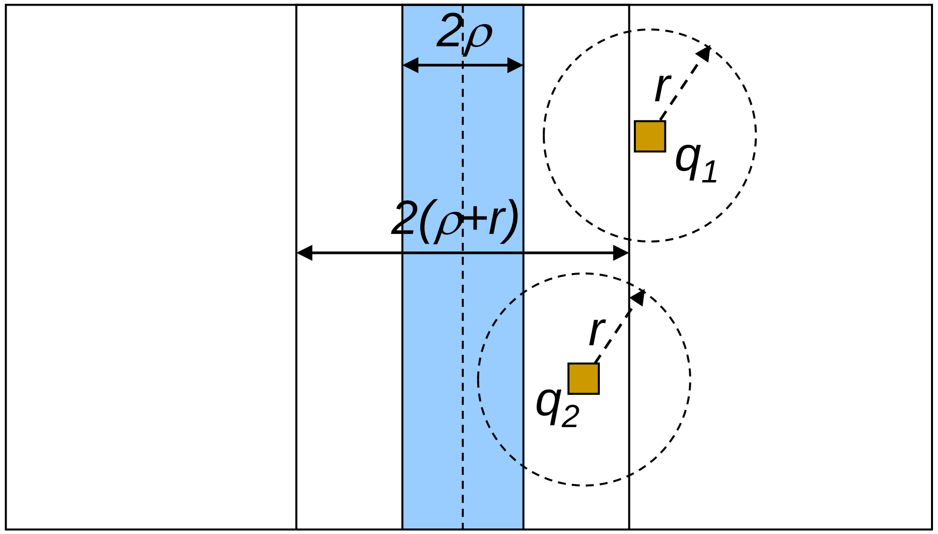
- Binary mapping: $bps^{1,\rho}: \mathcal{D} \rightarrow \{0,1,-\}$
  - ρ-split function, $\rho \geq 0$
  - also called the first order ρ-split function
- Separable property (up to 2ρ):

  $\forall x,y \in \mathcal{D},\ bps^{1,\rho}(x) = 0$ and $bps^{1,\rho}(y) = 1 \Rightarrow d(x,y) > 2\rho$
  - No objects closer than 2ρ can be found in both the separable sets.
- Symmetry property: $\forall x,y \in \mathcal{D},\ \rho_2 \geq \rho_1,$

  $bps^{1,\rho_2}(x) \neq -,\ bps^{1,\rho_1}(y) = - \quad \Rightarrow \quad d(x,y) > \rho_2 - \rho_1$

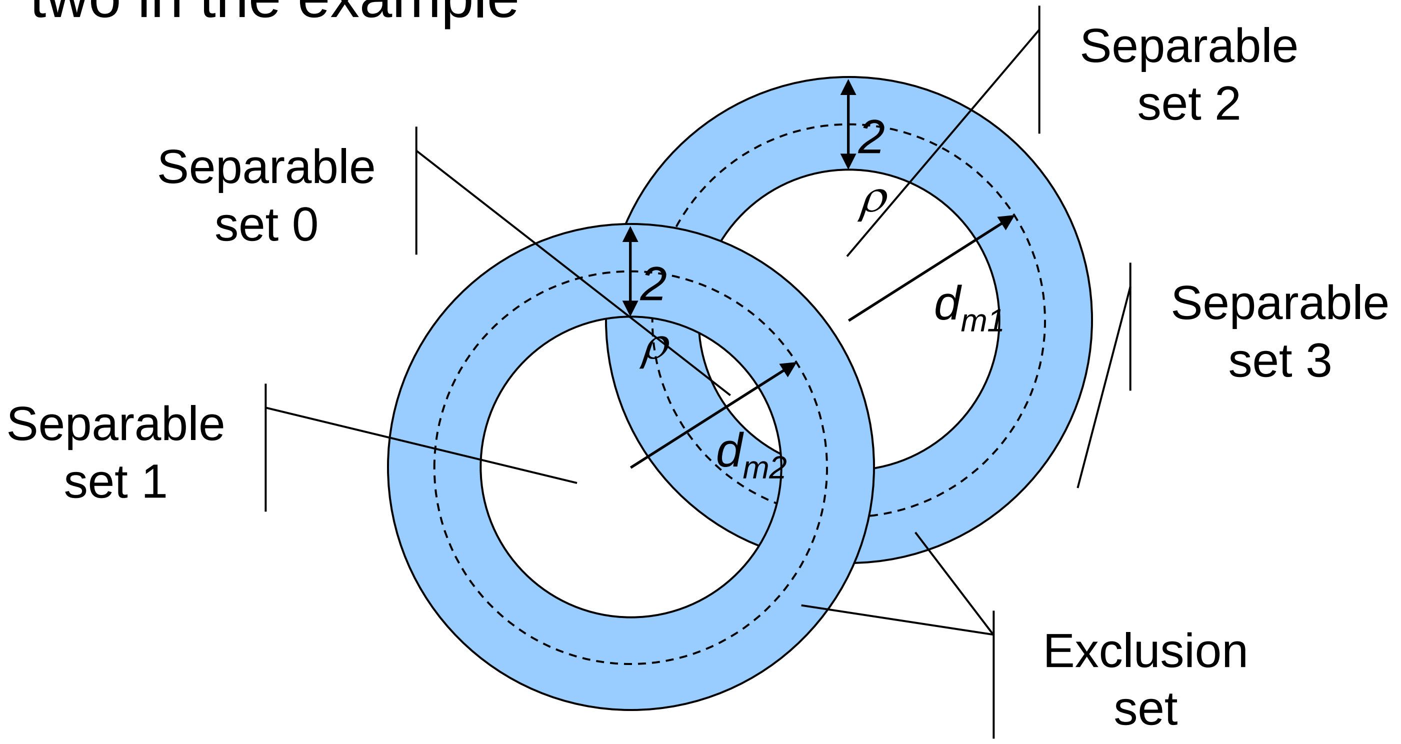
D-index: Symmetry Property

- Ensures that the exclusion set “shrinks” in a symmetric way as ρ decreases.
- We want to test whether a query intersects the exclusion set or not.



D-index: General ρ-Split Function

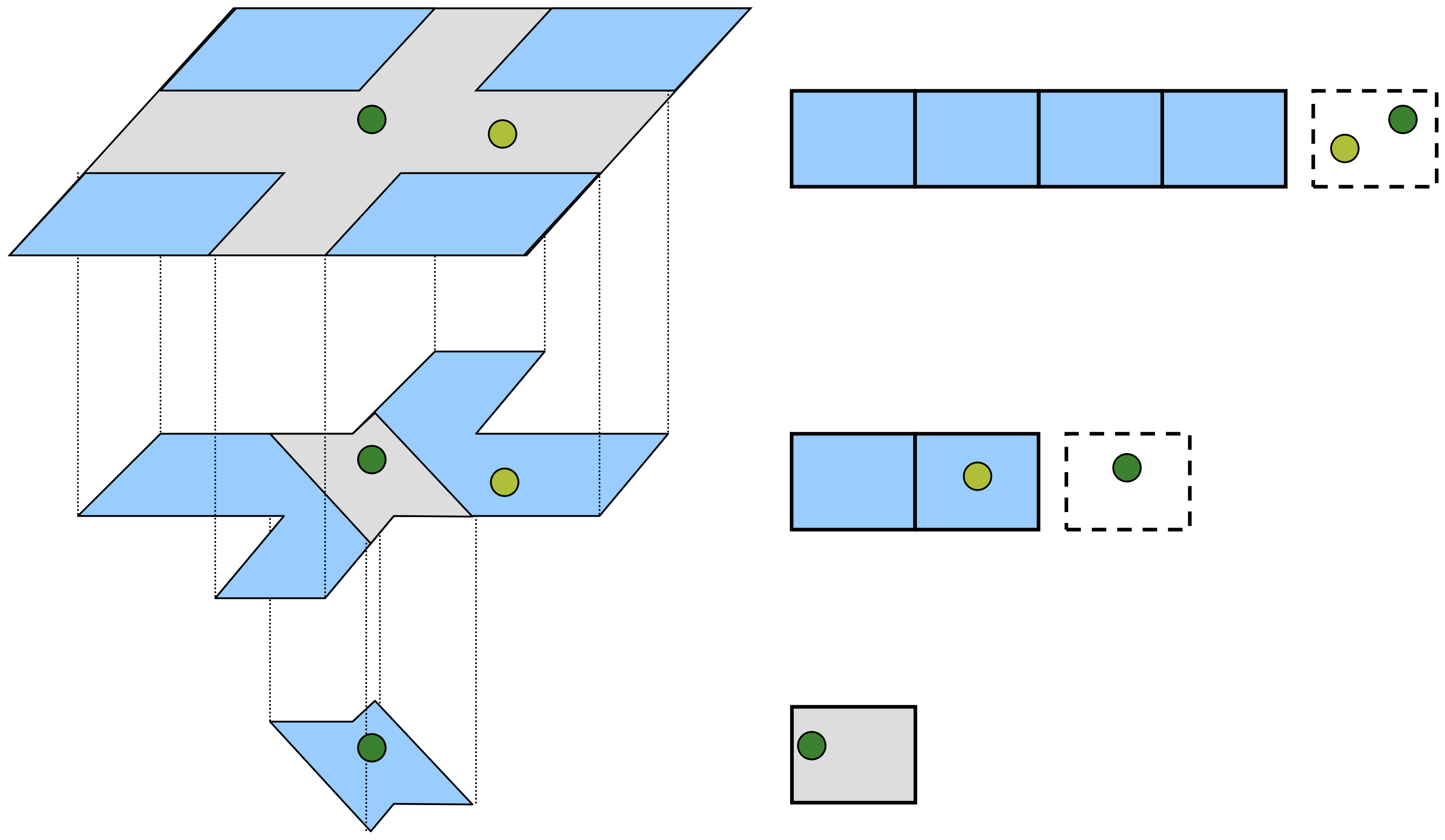
- Combination of several binary ρ-split functions
  - two in the example



D-index: General ρ-Split Function

- A combination of *n* first order ρ-split functions:
  - $bps^{n,\rho}: \mathcal{D} \rightarrow \{0..2^n-1, -\}$
  - $$bps^{n,\rho}(x) = \begin{cases} - & \text{if } \exists i, bps_i^{1,\rho}(x) = - \\ b & \text{all } bps_i^{1,\rho}(x) \text{ form a binary number } b \end{cases}$$
- Separable & symmetry properties hold
  - resulting sets are also separable up to 2ρ.

D-index: Insertion



D-index: Insertion Algorithm

- $Dindex^{\rho}(X, m_1, m_2, \ldots, m_h)$
  - h – number of levels,
  - m_i – number of binary functions combined on level i.
- Algorithm – insert the object o_N:

```
for $i$=$1$ to $h$ do
  if $bps^{m_i,\rho}(o_N) \neq$ '-' then
    $o_N \rightarrow$ bucket with the index $bps^{m_i,\rho}(o_N)$.
    exit
  end if
end do
$o_N \rightarrow$ global exclusion bucket.
```

D-index: Insertion Algorithm (cont.)

- The new object is inserted with one bucket access.
- Requires $\sum_{i=1}^{j} m_i$ distance computations
  - assuming o_N was inserted in a bucket on the level *j*.

D-index: Range Query

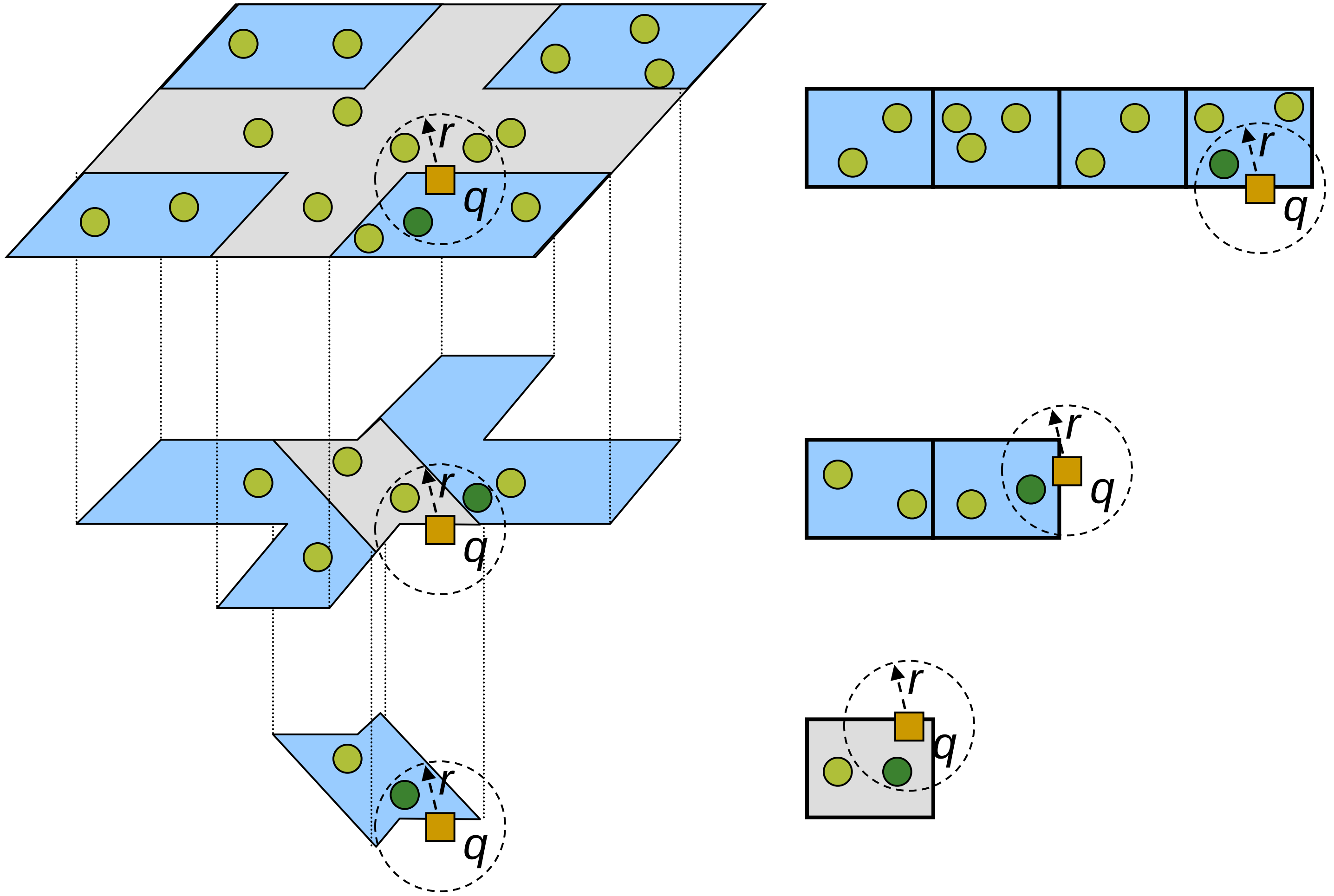
- $Dindex^{\rho}(X, m_1, m_2, \ldots, m_h)$
  - h – number of levels,
  - m_i – number of binary functions combined on level i.

Given a query $R(q,r)$ with $r \leq \rho$:

```
for $i=1$ to $h$ do
    search in the bucket with the index $bps^{m_i,0}(q)$.
end do
search in the global exclusion bucket.
```

- Objects o, $d(q,o) \leq r$, are reported on the output.

D-index: Range Search (cont.)



D-index: Range Query (cont.)

- The call $bps^{m_i,0}(q)$ always returns a value between 0 and $2^{m_i}-1$.
- Exactly one bucket per level is accessed if $r \leq \rho$
  - $h+1$ bucket access.
- Reducing the number of bucket accesses:
  - the query region is in the exclusion set $\Rightarrow$ proceed the next level directly,
  - the query region is in a separable set $\Rightarrow$ terminate the search.

D-index: Advanced Range Query

```
for $i = 1$ to $h$
    if $bps^{m_i,\rho+r}(q) \neq -$ then          (exclusively in the separable bucket)
        search in the bucket with the index $bps^{m_i,\rho+r}(q)$.
        exit                                    (search terminates)
    end if
    if $r \leq \rho$ then                         (the search radius up to $\rho$)
        if $bps^{m_i,\rho-r}(q) \neq -$ then      (not exclusively in the exclusion zone)
            search in the bucket with the index $bps^{m_i,\rho-r}(q)$.
        end if
    else                                        (the search radius greater than $\rho$)
        let $\{i_1,\ldots i_n\} = G(bps^{m_i,r-\rho}(q))$
        search in the buckets with the indexes $i_1,\ldots,i_n$.
    end if
end for
search in the global exclusion bucket.
```

D-index: Advanced Range Query (cont.)

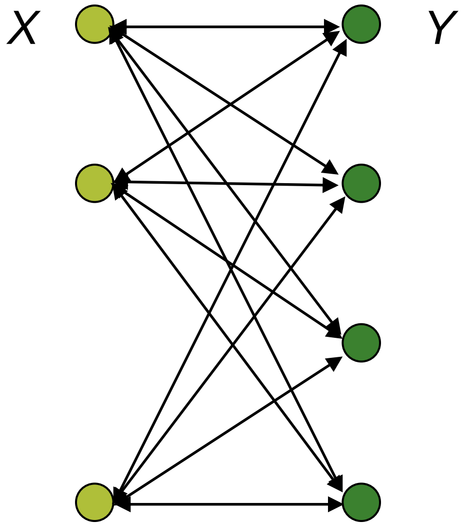
- The advanced algorithm is not limited to $r \leq \rho$.
- All tests for avoiding some bucket accesses are based on manipulation of parameters of split functions (i.e. ρ).
- The function *G*() returns a set of bucket indexes:
  - all minuses (-) in the split functions' results are substituted by all combinations of ones and zeros,
  - e.g. $bps^{3,\rho}(q)$=*'1--'*
  - $G(bps^{3,\rho}(q))=\{100,101,110,111\}$

D-index: Features

- supports disk storage
- insertion needs one bucket access
  - distance computations vary from m_1 up to $\sum_{i=1..h} m_i$
- $h+1$ bucket accesses at maximum
  - for all queries such that qualifying objects are within ρ
- exact match ($R(q,0)$)
  - successful – one bucket access
  - unsuccessful – typically no bucket is accessed

Similarity Join Query

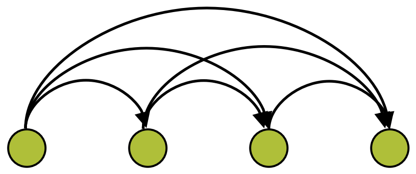
- The similarity join can be evaluated by a simple algorithm which computes |*X*||*Y*| distances between all the pairs of objects.



= *NM* distance computations

Similarity Self Join Query

- The similarity *self* join examines all pairs of objects of a set *X*, which is |*X*||*X*| distance computations.
- Due to the symmetry property, *d(x,y)* = *d(y,x)*, we can reduce the costs.

X  $= \frac{N(N-1)}{2}$ distance computations

- This is called the *nested loops algorithm* (*NL*).

Similarity Self Join Query (cont.)

- Specialized algorithms
  - usually built on top of a commercial DB system, or
  - tailored to specific needs of application.
- D-index provides a very efficient algorithm for range queries:
  - a self join query can be evaluated using

Range Join Algorithm (*RJ*)*:*

```
for each o in dataset X do
    range_query(o, μ)
end do
```

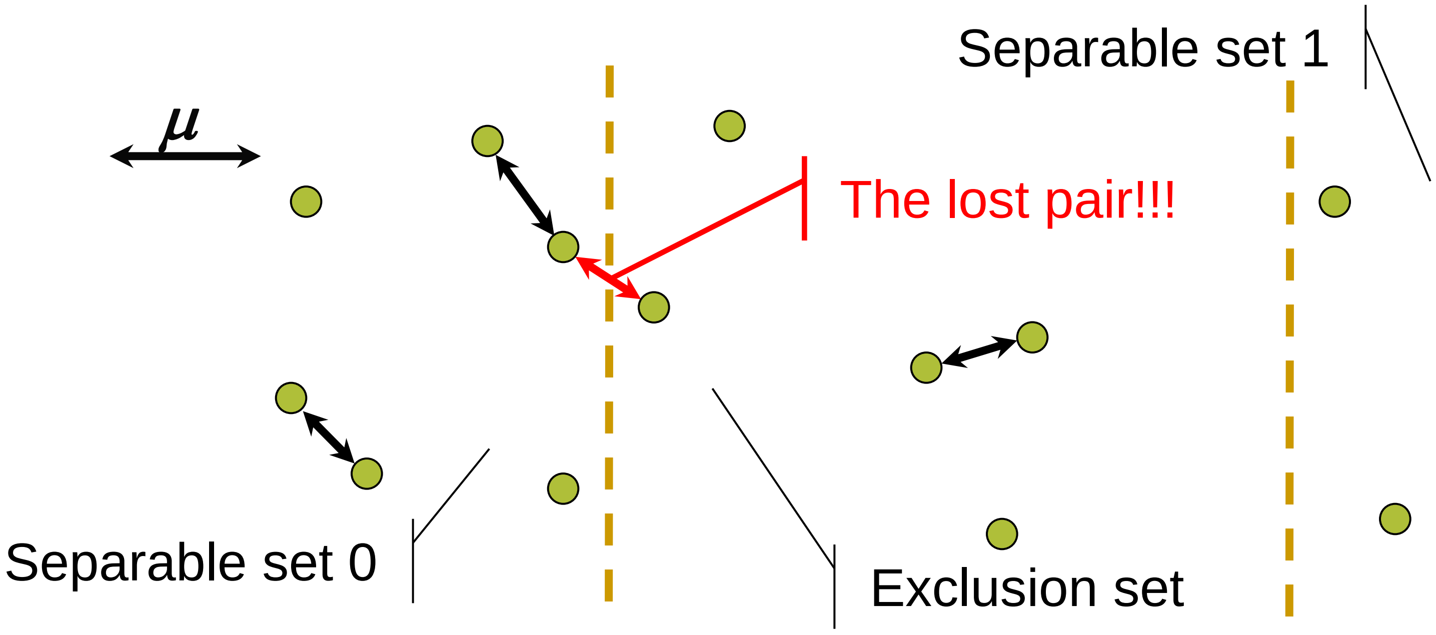
Extended D-index (eD-index)

- A variant of D-index which provides a specialized algorithm for similarity joins.
- Application independent – general solution.

- Split functions manage replication.
- D-index’s algorithms for range & *k-NN* queries are only slightly modified.

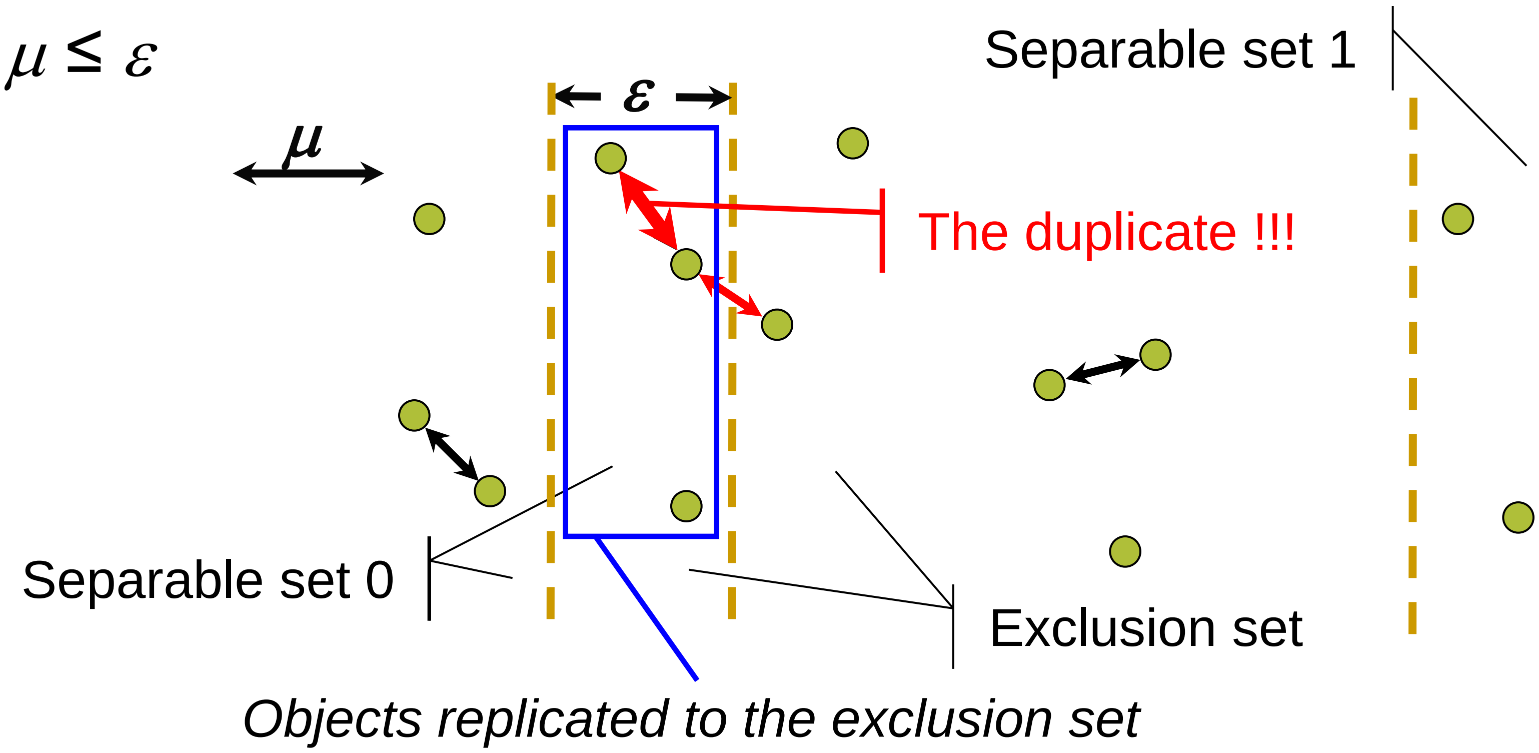
eD-index: Similarity Self Join Query

- Similarity self join is elaborated independently in each bucket.
- The result set is a union of answers of all sub-queries.

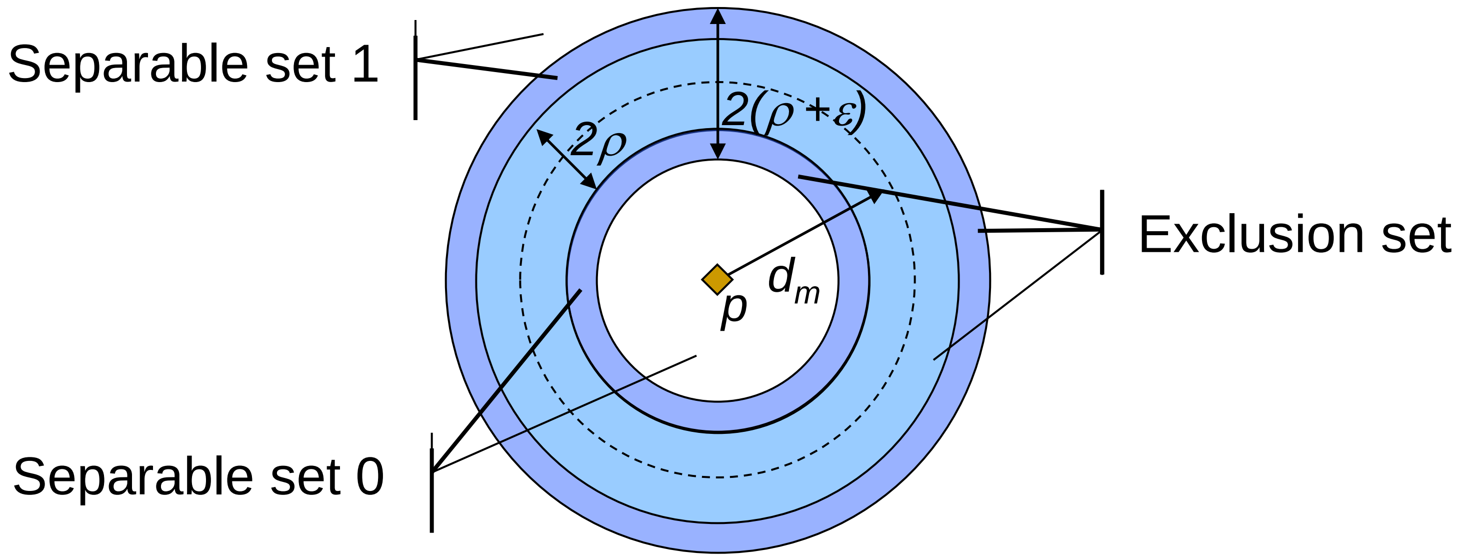


eD-index: Overloading Principle

- Lost pairs are handled by replications
  - areas of width ε are replicated in the exclusion set.
- $\mu \leq \varepsilon$



eD-index: ρ-Split Function Modification



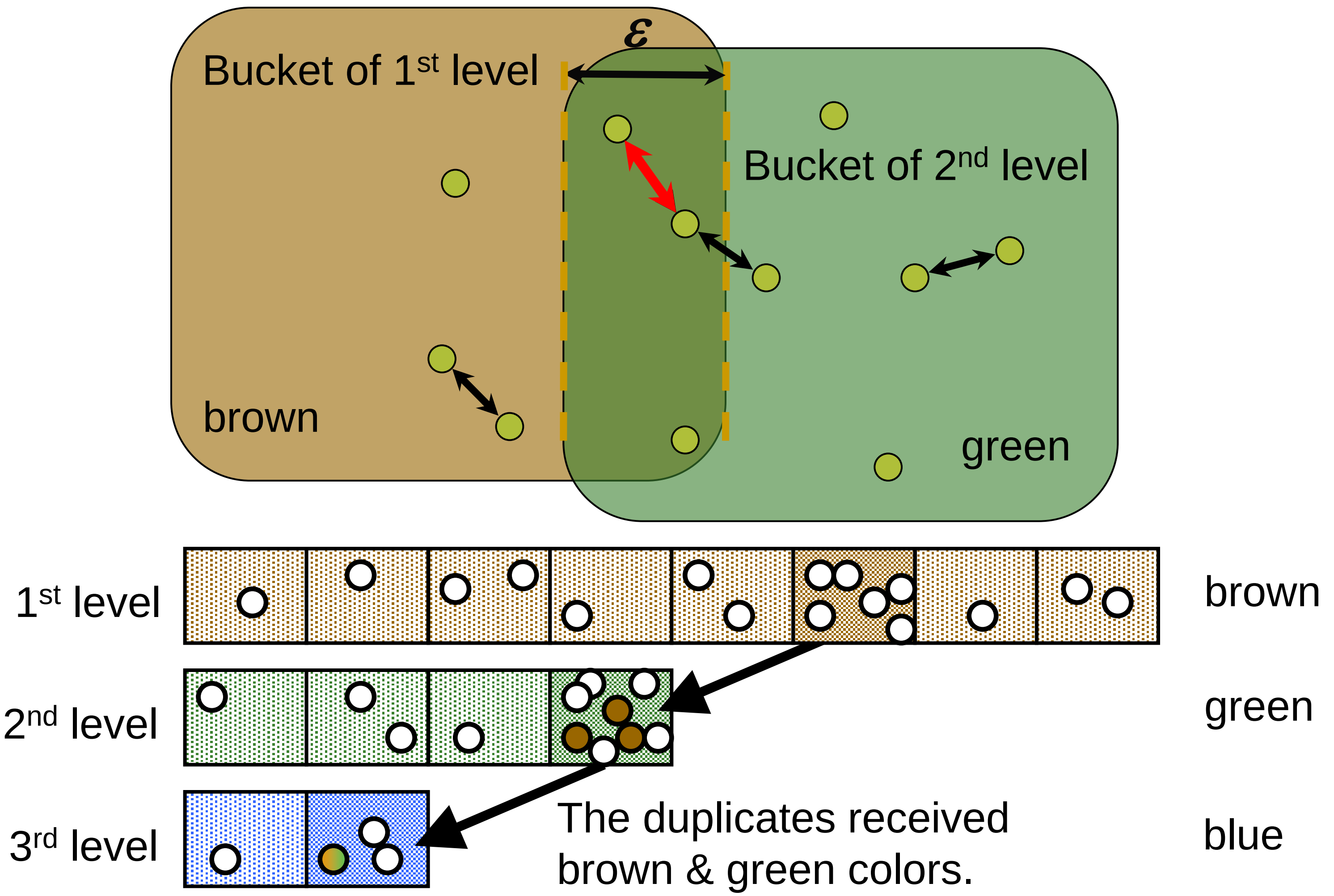
- The modification of ρ-split function is implemented in the insertion algorithm by varying the parameter ρ
  - the original stop condition in the D-index's algorithm is changed.

eD-index: Insertion Algorithm

- $eDindex^{\rho,\varepsilon}(X, m_1, m_2, \ldots, m_h)$
- Algorithm – insert the object o_N:

```
for $i=1$ to $h$ do
  if $bps^{m_i,\rho}(o_N) \neq$ '-' then
    $o_N \rightarrow$ bucket with the index $bps^{m_i,\rho}(o_N)$.
    if $bps^{m_i,\rho+\varepsilon}(o_N) \neq$ '-' then      (not in the overloading area)
      exit
    end if
  end if
end do
$o_N \rightarrow$ global exclusion bucket.
```

eD-index: Handling Duplicates



The duplicates received brown & green colors.

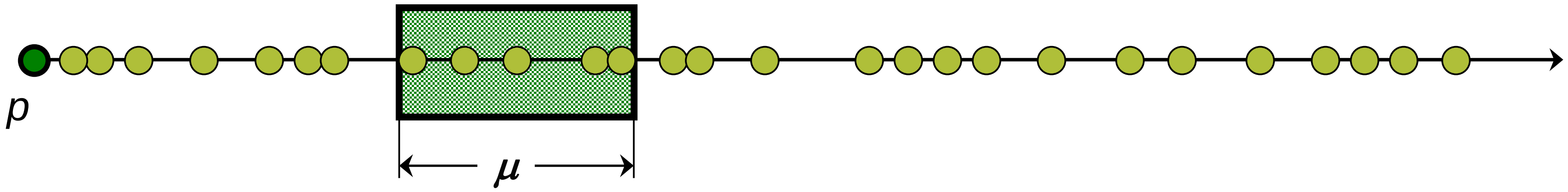
eD-index: Overloading Join Algorithm

Given similarity self-join query *$SJ(\mu)$:*

- Execute the query in every separable bucket on every level
  - and in the global exclusion bucket.
- In the bucket, apply *sliding window* algorithm.
- The query's result is formed by concatenation of all sub-results.

eD-index: Sliding Window

- Use the triangle inequality
  - to avoid checking all pairs of objects in the bucket.
- Order all objects on distances to one pivot.
- The sliding window is then moved over all objects.
  - only pairs of objects in the window are examined.



- Due to the triangle inequality, the pair of objects outside the window cannot qualify:
  - $d(x,y) \geq d(x,p) - d(y,p) > \mu$

eD-index: Sliding Window (cont.)

- The algorithm also employs
  - the pivot filtering and
  - the eD-index's coloring technique.
- Given a pair of objects o_1,o_2:
  - if a color is shared, this pair must have been reported on the level having this color – the pair is ignored without distance computation, else
  - if $d(o_1,o_2)\leq\mu$, it is an original qualifying pair.

eD-index: Limitations

- Similarity self-join queries only
  - the query selectivity must satisfy: $\mu \leq \varepsilon$.
  - it is not very restrictive since we usually look for close pairs.
- The parameters ρ and ε depend on each other.
  - $\varepsilon \leq 2\rho$
  - If $\varepsilon > 2\rho$, the overloading zone is wider than the exclusion zone.
    - because we do not replicate objects between separable sets – only between a separable set and the exclusion zone,
    - some qualifying pairs might be missed.

Centralized Index Structures for Large Databases

1. M-tree family
2. hash-based metric indexing
3. **performance trials**

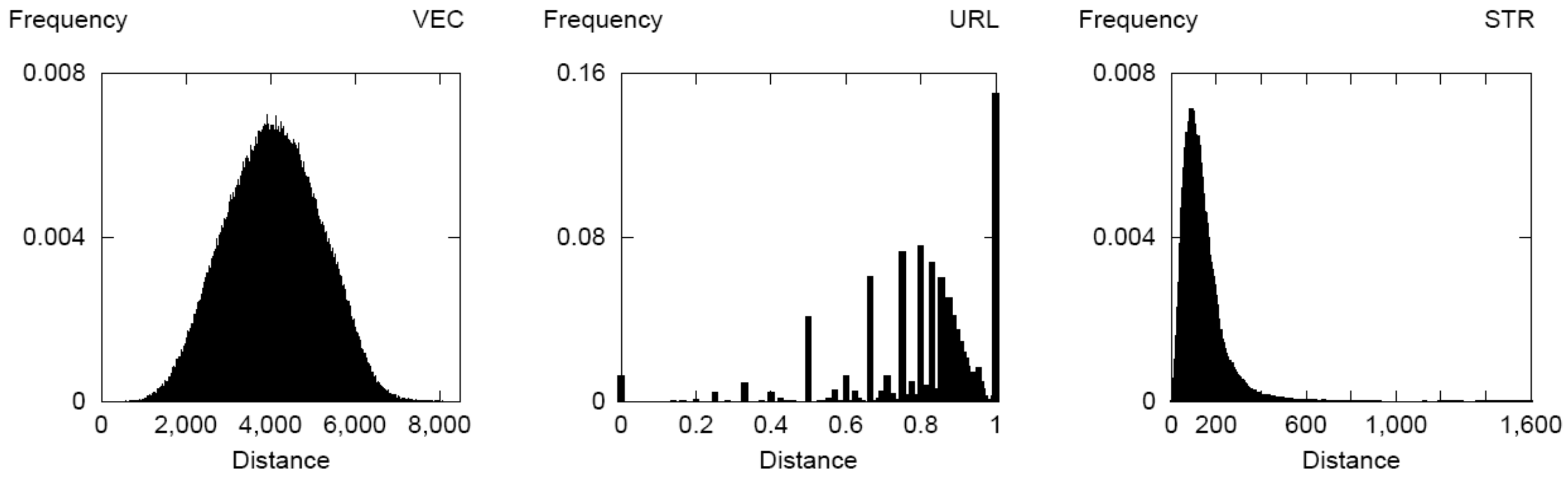
Performance Trials

- experiments on M-tree and D-index
- three sets of experiments:
  1. **comparison** of M-tree (tree-based approach) vs. D-index (hash-based approach)
  2. processing different **types of queries**
  3. **scalability** of the centralized indexes – growing the size of indexed dataset

Datasets and Distance Measures

- trials performed on three datasets:
  - **VEC:** 45-dimensional vectors of image color features compared by the *quadratic distance* measure
  - **URL:** sets of URL addresses; the distance measure is based on the similarity of sets (*Jaccard’s coefficient*)
  - **STR:** sentences of a Czech language corpus compared using an *edit distance*

Datasets: Distance Distribution



- distribution of distances within the datasets:
  - VEC: practically normal distance distribution
  - URL: discrete distribution
  - STR: skewed distribution

Trials: Measurements & Settings

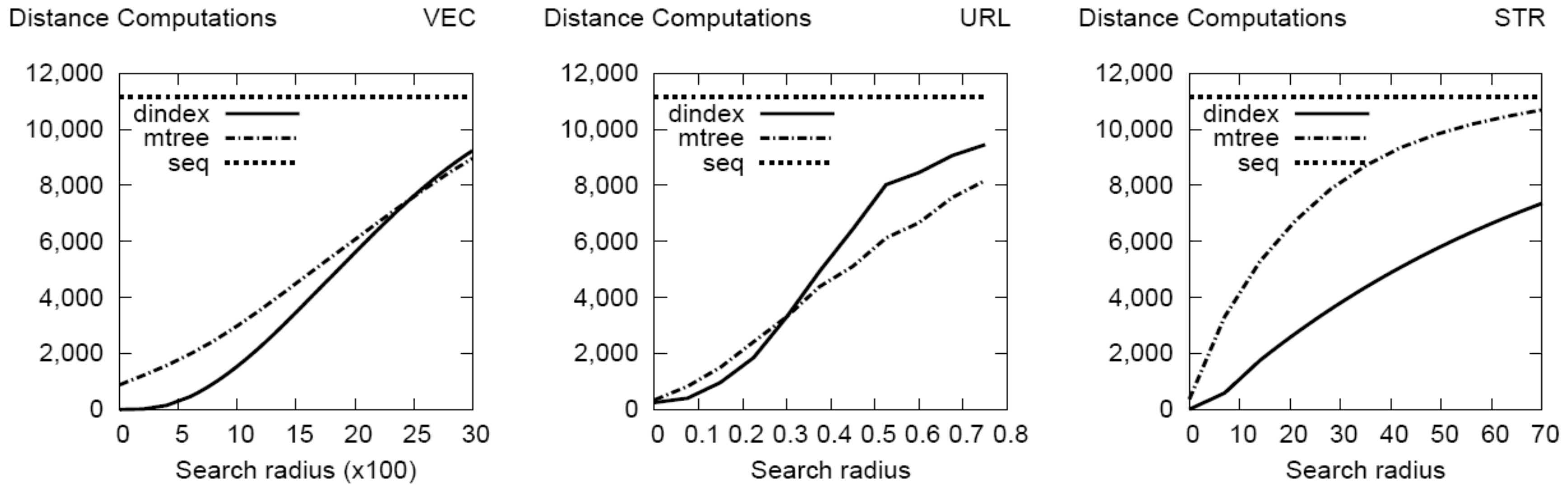
- CPU costs: number of distance computations
- I/O costs: number of block reads
  - The same size of disk blocks

- Query objects follow the dataset distribution
- Average values over 50 queries:
  - Different query objects
  - The same selectivity
    - Radius or number of nearest neighbors

Comparison of Indexes

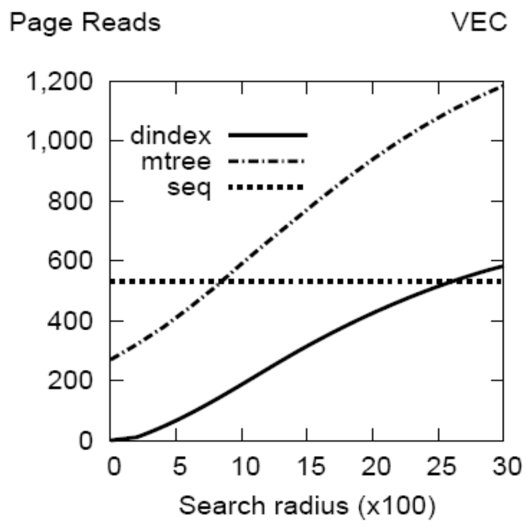
- Comparing performance of
  - M-tree – a tree-based approach
  - D-index – hash-based approach
  - *sequential scan* (baseline)
- Dataset of 11,100 objects
- Range queries – increasing radius
  - maximal selectivity about 20% of the dataset

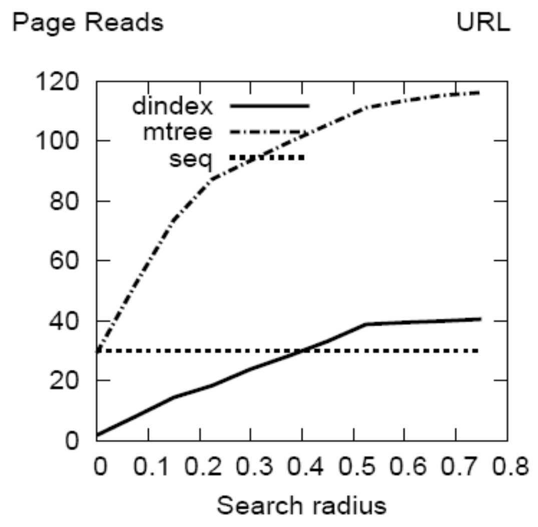
Comparison: CPU Costs

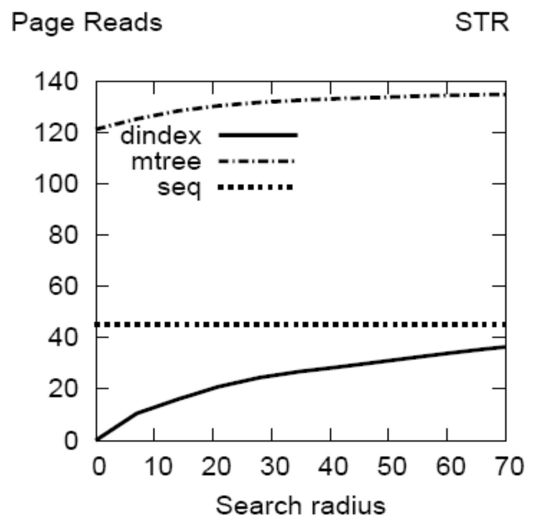


- generally, D-index outperforms M-tree for smaller radii
- D-index: pivot-based filtering depends on data distribution and query size
- M-tree outperforms D-index for discrete distribution
  - pivot selection is more difficult for discrete distributions

Comparison: I/O Costs





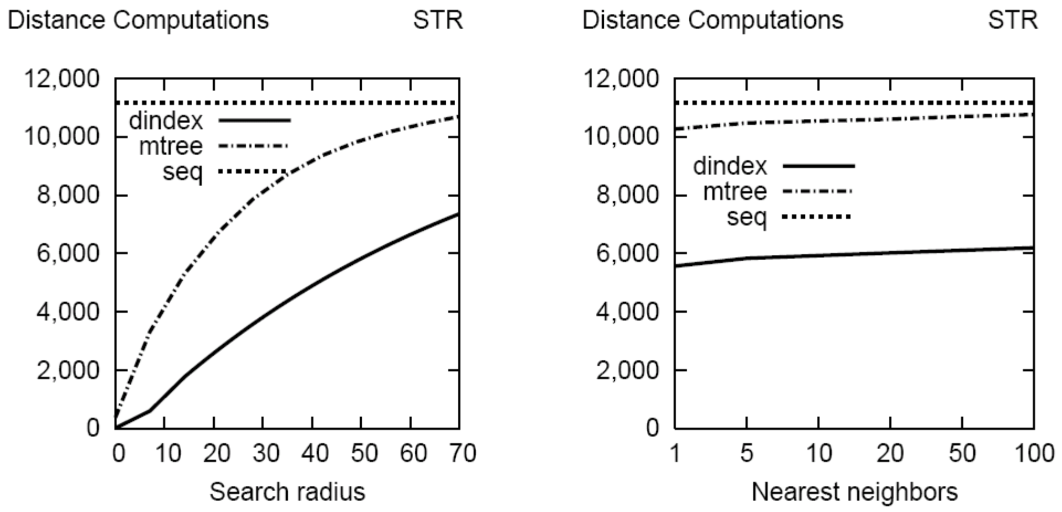


- M-tree needs twice the disk space to stored data than SEQ
- inefficient if the *distance function* is easy to compute
- D-index more efficient
- a query with *r=0*: D-index accesses only one page (important, e.g., for deletion)

Different Query Types

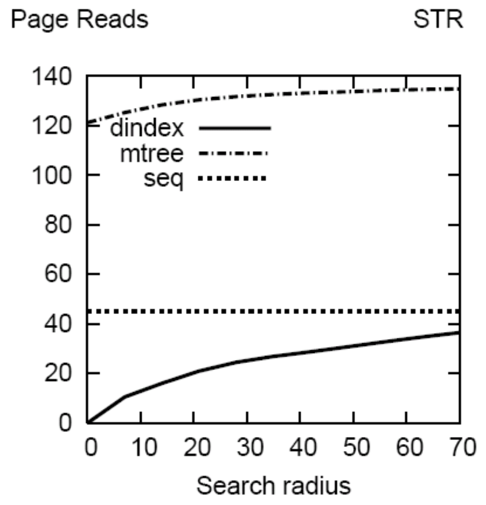
- comparing processing performance of different types of queries
  - range query
  - nearest neighbor query
  - similarity self join
- M-tree, D-index, sequential scan

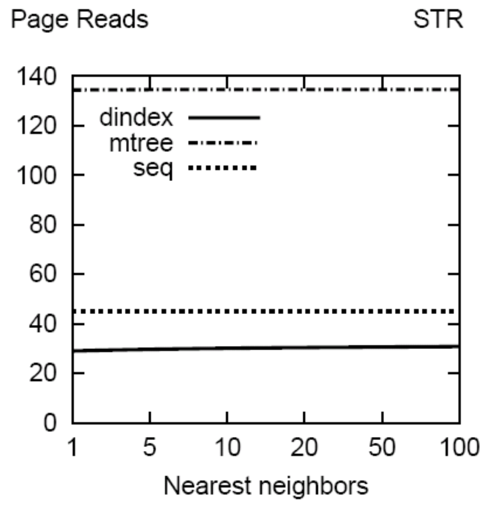
Range vs. *k*-NN: CPU Costs



- nearest neighbor query:
  - similar trends for M-tree and D-index
  - the D-index advantage of small radii processing decreases
  - expensive even for small *k* – similar costs for both 1 and 100
  - D-index still twice as fast as M-tree

Range vs. *k*-NN: I/O Costs





- nearest neighbor query:
  - similar trends for I/O costs as for CPU costs
  - D-index four times faster than M-tree

Similarity Self Join: Settings

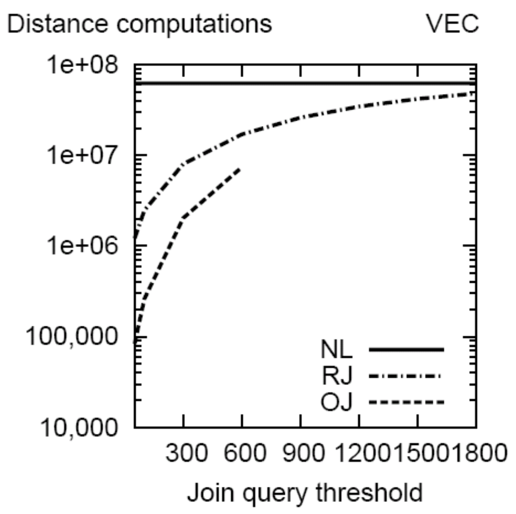
- $J(X,X,\mu)$ – very demanding operation
- three algorithms to compare:
  - NL: nested loops – naive approach
  - RJ: range join – based on D-index
  - OJ: overloading join – eD-index
    - for μ: $2\mu \leq \rho$, i.e. $\mu \leq 600$ for vectors
- datasets of about 11,000 objects
- selectivity – retrieving up to 1,000,000 pairs (for high values of μ)

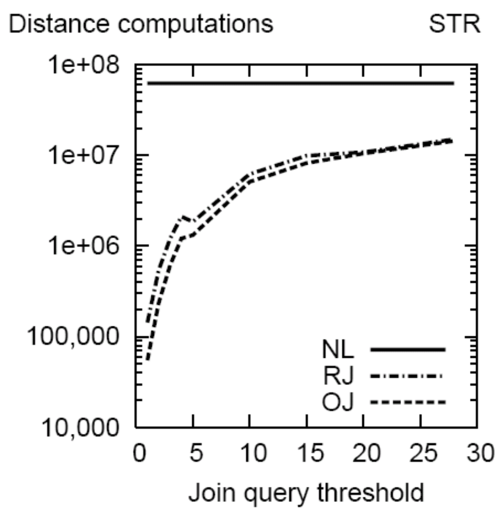
Similarity Self Join: Complexity

- Quadratic complexity
  - prohibitive for large DB
  - example: 50,000 sentences
  - a range query:
    - sequential scan takes about 16 seconds

  - a self join query:
    - nested loops algorithm takes 25,000 times more
    - about 4 days and 15 hours!

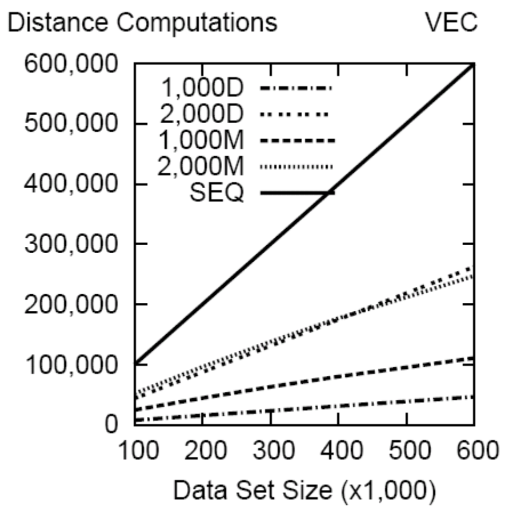
Similarity Join: Results

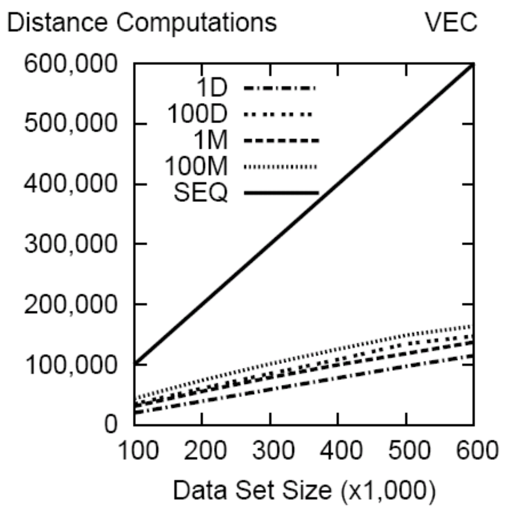




- RJ and OJ costs increase rapidly (logarithmic scale)
- OJ outperforms RJ twice (STR) and 7 times for VEC:
  - high distances between VEC objects
  - high pruning effectiveness of pivot-based filtering for smaller μ

Scalability: CPU Costs

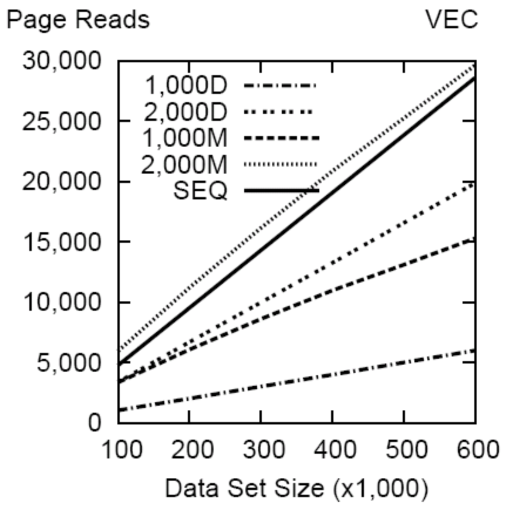


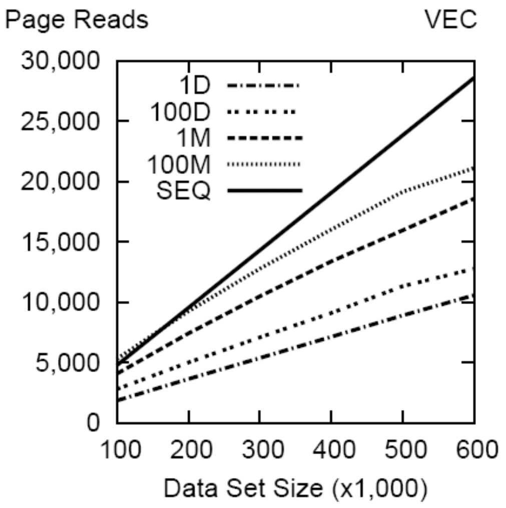


- range query: *r* = 1,000; 2,000
- *k-NN* query: *k* = 1; 100

- labels: radius or *k* + D (D-index), M (M-tree), SEQ
- data: from 100,000 to 600,000 objects
- M-tree and D-index are faster (D-index slightly better)
- linear trends

Scalability: I/O Costs





- the same trends as for CPU costs
- D-index more efficient than M-tree
- *exact match* contrast:
  - M-tree: 6,000 block reads + 20,000 d. c. for 600,000 objects
  - D-index: read 1 block + 18 d. c. regardless of the data size

Scalability: Similarity Self Join

- We use the *speedup s* as the performance measure:

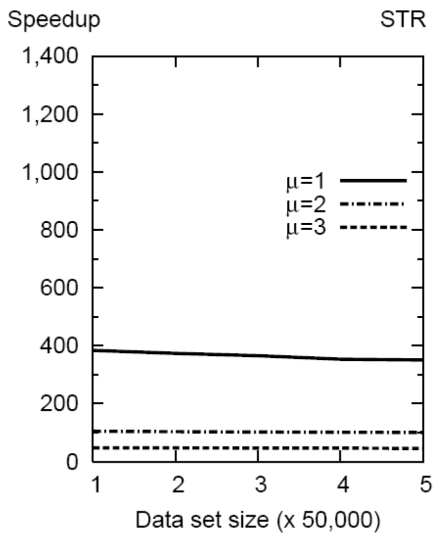
$$s = \frac{\boxed{N(N-1)}}{2\boxed{n}}$$

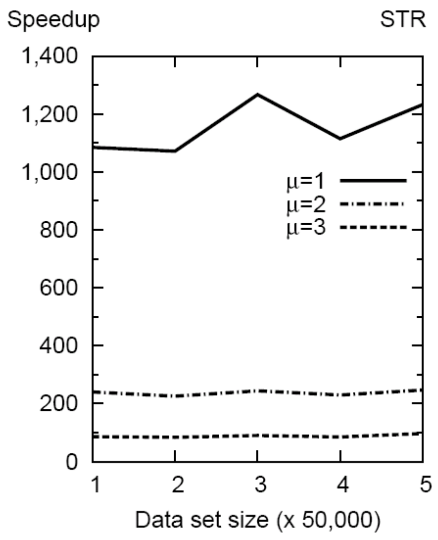
Distance computations of Nested Loops

An algorithm's distance computations

- Speedup measures how many times is a specific algorithm faster than NL.

Scalability: Similarity Self Join (cont.)





- RJ: range join
- OJ: overloading join

- STR dataset: from 50,000 to 250,000 sentences
- constant speedup
  - E.g. a join query on 100,000 objects takes 10 minutes.
  - The same join query on 200,000 objects takes 40 minutes.
- OJ at least twice faster than RJ

Scalability Experiments: Conclusions

- similarity search is expensive
- the scalability of centralized indexes is linear

- cannot be applied to huge data archives
  - become inefficient after a certain point

Possible solutions:

- sacrifice some precision: **approximate techniques**
- use more storage & computational power: **distributed data structures**