# Indexing Multimedia Data

Tecnologie delle Basi di Dati M

# Plan of activities

- In the following we will go through 2 distinct topics, all of them being related by the common objective to provide efficient support to the execution of MM similarity queries

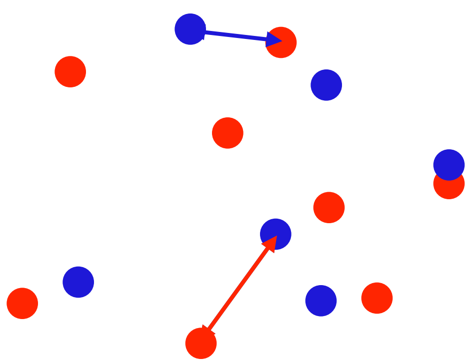
1. We will first consider metric trees, which allow us to deal even with non-vector features and with distance functions other than (weighted) Lp-norms
2. Finally, we will try to shed some light on the phenomenon of dimensionality curse, and then present some index structures that have been designed to (partially) solve such problem

# Beyond vector spaces

- It's a matter of fact that vector spaces, equipped with some (weighted) Lp-norm, are not general enough to deal with the whole variety of feature types and distance functions needed for MM data

Example:

given 2 sets of points s1 and s2, their Hausdorff distance is defined as follows:



1 ∀ (red) point of s1 find the closest (blue) point in s2
Let **h(s1,s2)** be the maximum of such distances

2 ∀ (blue) point in s2 find the closest (red) point in s1
Let **h(s2,s1)** be the maximum of such distances

3 Let $d_{Haus}(s1,s2) = \max\{ h(s1,s2), h(s2,s1) \}$

Used for matching shapes

# Another example: set similarity

- We have logs of WWW accesses, where each log entry has a format like:

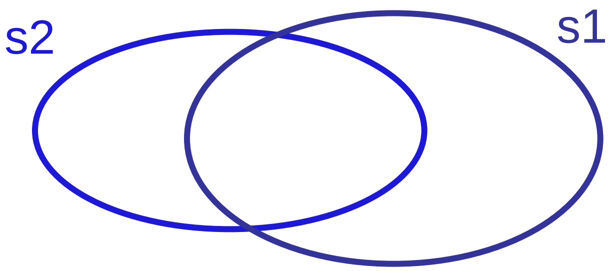
```
www-db.deis.unibo.it pciaccia -
        [11/Jan/1999:10:41:37 +0100]
        "GET /~mpatella/ HTTP/1.0" 200 1573
```

- Log entries are grouped into sessions (= sets of visited pages):

  S = <ip_address, user_id, [$url_1$,…,$url_k$]>

  and we want to compare "similar sessions" (i.e., similar sets), using:

$$d_{setdiff}(s1,s2) = \frac{|s1 - s2| + |s2 - s1|}{|s1| + |s2|}$$



# Another example: edit distance

- A common distance measure for strings is the so-called edit distance, defined as the minimum number of characters that have to be inserted, deleted, or substituted so as to transform a string s1 into another string s2

$d_{edit}$('ball','bull') = 1 $d_{edit}$('balls','bell') = 2 $d_{edit}$('rather','alter') = 3

- The edit distance is also commonly used in genomic DB's to compare DNA sequences.
- Each DNA sequence is a string over the 4-letters alphabet of bases:

a: adenine

c: cytosine

g: guanine

t: thymine

$d_{edit}$('gatctggtgg','agcaaatcag') = 7

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| g | a | t | c | t | g | g | t | g | - | g |
| 1 | = | 2 | = | 3 | 4 | 5 | = | 6 | 7 | = |
| - | a | g | c | a | a | a | t | c | a | g |

The edit distance can be computed using a dynamic programming procedure, similar to the one seen for the DTW

# Computing the Edit Distance

- The cost matrix is used to incrementally build the new matrix $d_{edit}$, whose elements are recursively defined as:

$$d_{edit;i,j} = cost_{i,j} + \min\{d_{edit;i-1,j}, d_{edit;i,j-1}, d_{edit;i-1,j-1}\}$$

s2 ↑

| | | | | | | |
|---|---|---|---|---|---|---|
| r | 1 | 1 | 1 | 1 | 1 | 0 |
| e | 1 | 1 | 1 | 1 | 0 | 1 |
| h | 1 | 1 | 1 | 1 | 1 | 1 |
| t | 1 | 1 | 1 | 0 | 1 | 1 |
| a | 1 | 0 | 1 | 1 | 1 | 1 |
| r | 1 | 1 | 1 | 1 | 1 | 0 |
| | 0 | 1 | 1 | 1 | 1 | 1 |
| cost | | a | l | t | e | r |

→ s1

s2 ↑

| | | | | | | |
|---|---|---|---|---|---|---|
| r | 6 | 5 | 5 | 5 | 4 | **3** |
| e | 5 | 4 | 4 | 4 | **3** | 4 |
| h | 4 | 3 | 3 | **3** | 3 | 4 |
| t | 3 | 2 | 3 | **2** | 3 | 4 |
| a | 2 | **1** | **2** | 3 | 4 | 5 |
| r | **1** | 1 | 2 | 3 | 4 | 4 |
| | **0** | 1 | 2 | 3 | 4 | 5 |
| $d_{edit}$ | | a | l | t | e | r |

→ s1

# Metric spaces

- A metric space **M = (U,d)** is a pair, where
  U is a domain ("universe") of values, and
  d is a distance function that, $\forall$ x,y,z $\in$ U, satisfies the metric axioms:

| | |
|---|---|
| **$d(x,y) \geq 0$, $d(x,y) = 0 \Leftrightarrow x = y$** | **(positivity)** |
| **$d(x,y) = d(y,x)$** | **(symmetry)** |
| **$d(x,y) \leq d(x,z) + d(z,y)$** | **(triangle inequality)** |

  - All the distance functions seen in the previous examples are metrics, and so are the (weighted) Lp-norms
  - The only distance we have seen so far that does not fit the metric framework is the DTW

Metric indexes only use the metric axioms
to organize objects, and exploit
the triangle inequality to prune the search space

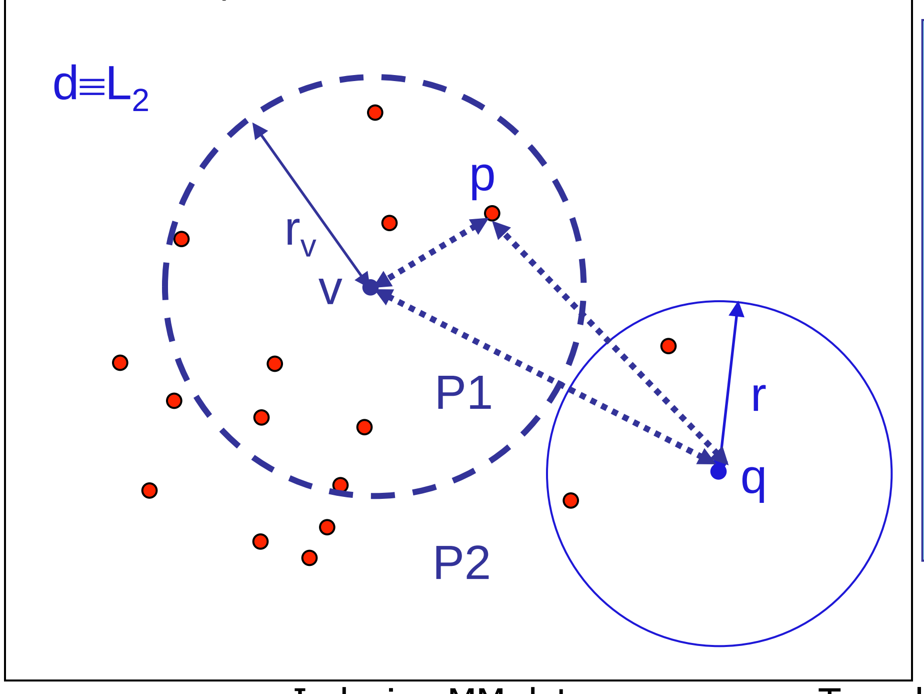
# Principles of metric indexing (1)

- Given a "metric dataset" $P \subseteq \mathbf{U}$, one of the two following principles can be applied to partition it into two subsets

Ball decomposition: take a point v ("vantage point"), compute the distances of all other points p w.r.t. v, d(p,v), and define

$$P1 = \{p : d(p,v) \leq r_v\} \qquad P2 = \{p : d(p,v) > r_v\}$$

If $r_v$ is chosen so that $|P1| \approx |P2| \approx |P|/2$ we obtain a balanced partition




Consider a range query $\{p: d(p,q) \leq r\}$

If $\mathbf{d(q,v) > r_v + r}$ we can conclude that no point in P1 belongs to the result

**Proof**:

we show that $d(p,q) > r$ holds $\forall p \in P1$.

$$\begin{aligned} d(p,q) &\geq d(q,v) - d(p,v) && \text{(triangle ineq.)} \\ &> r_v + r - d(p,v) && \text{(by hyp.)} \\ &\geq r_v + r - r_v && \text{(by def. of P1)} \\ &\geq r \end{aligned}$$

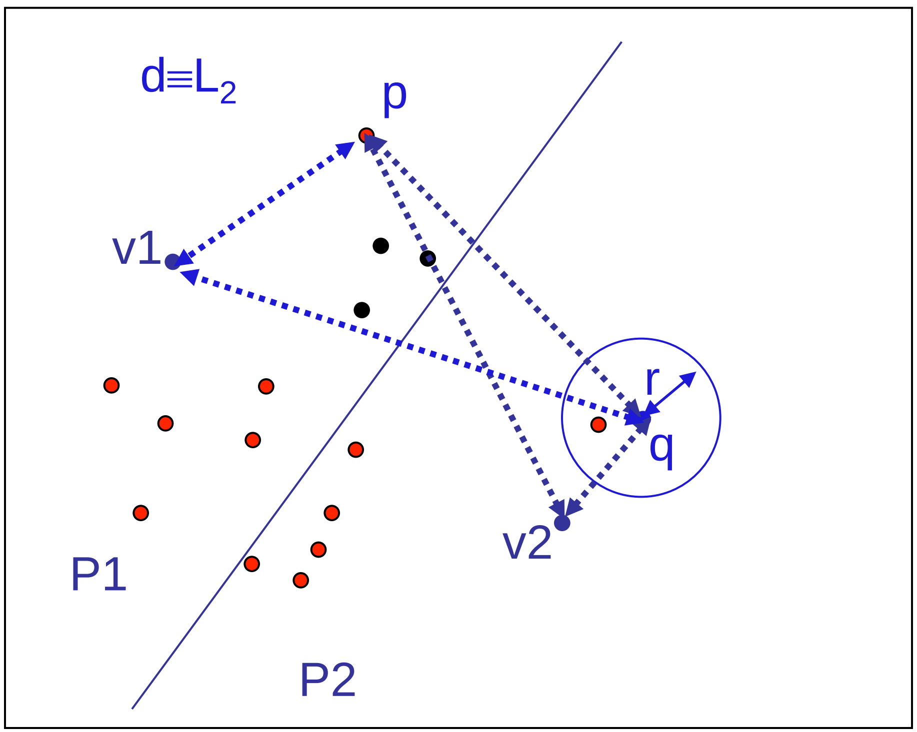
■

Similar arguments can be applied to P2

# Principles of metric indexing (2)

Generalized Hyperplane: take two points v1 and v2, compute the distances of all other points p w.r.t. v1 and v2, and define

$$P1 = \{p : d(p,v1) \leq d(p,v2)\} \quad P2 = \{p : d(p,v2) < d(p,v1)\}$$




Consider a range query $\{p: d(p,q) \leq r\}$

If **$d(q,v1) - d(q,v2) > 2*r$** we can conclude that no point in P1 belongs to the result

**Proof**:

we show that $d(p,q) > r$ holds $\forall p \in P1$.

$d(q,v1) - d(p,q) \leq d(p,v1)$ (triangle ineq.)

$d(p,v1) \leq d(p,v2)$ (def. of P1)

$d(p,v2) \leq d(p,q) + d(q,v2)$ (triangle ineq.)

Then:

$d(q,v1) - d(p,q) \leq d(p,q) + d(q,v2)$

$d(p,q) \geq (d(q,v1) - d(q,v2))/2$

$> r$ (by hyp.) ■

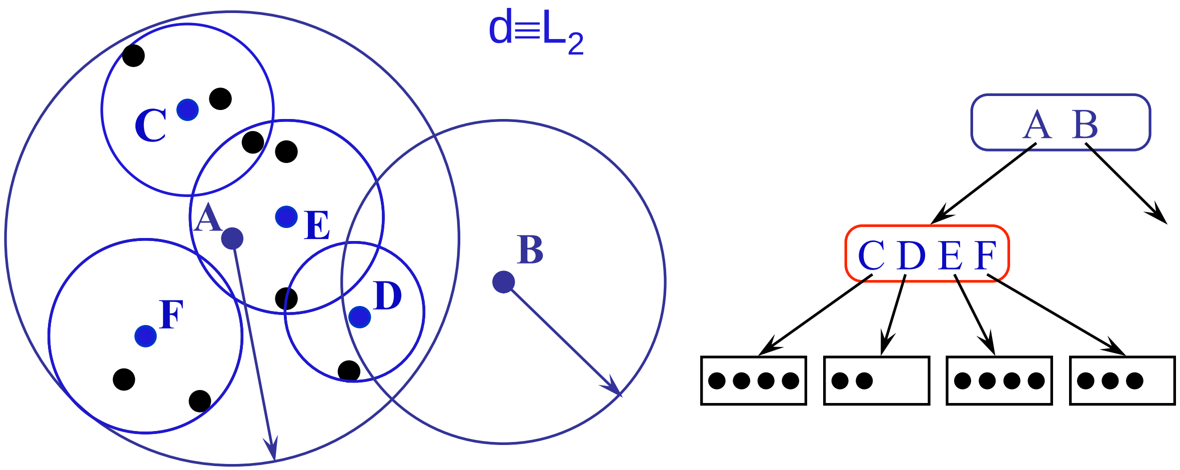
# The M-tree (Ciaccia, Patella & Zezula, 1997)

- The M-tree has been the first dynamic, paged, and balanced metric index
- Intuitively, it generalizes “R-tree principles” to arbitrary metric spaces
  - The M-tree treats the distance function as a “black box”
- Since 1997 [CPZ97], the M-tree has been used by several research groups for:
  - Image retrieval, text indexing, shape matching, clustering algorithms, fingerprint matching, DNA DB’s, etc.
  - [CNB+01] and [HS03] are both excellent surveys on searching in metric spaces
- C++ source code freely available at http://www-db.deis.unibo.it/Mtree/

**Remind**: at a first sight, the M-tree “looks like” an R-tree.

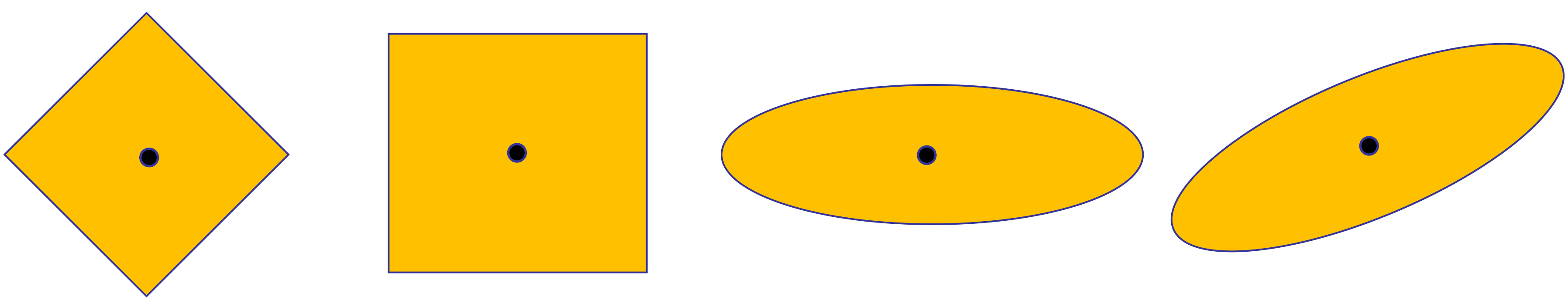
However, remember that the M-tree only “knows” about distance values, thus it ignores coordinate values and does not rely on any “geometric” (coordinate-based) reasoning

# M-tree: how it looks like




- Recursive bottom-up aggregation of objects based on regions
- Regions can overlap
- Each node can contain up to C entries, but not less than c ≤ 0.5*C
  - The root makes an exception

- Depending on the metric, the “shape” of index regions changes



$L_1$ $L_\infty$ Weighted Euclidean quadratic distance

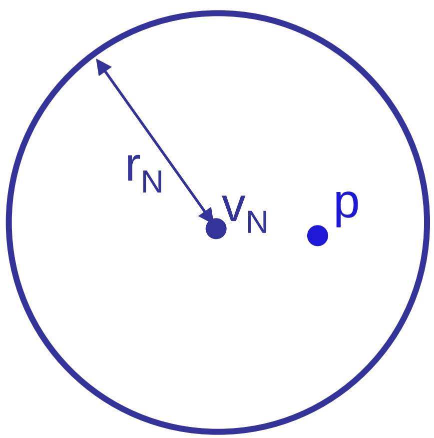
# The M-tree regions

- Each node N of the tree has an associated region, Reg(N), defined as

  $$\mathbf{Reg(N) = \{p: p \in U , d(p,v_N) \leq r_N\}}$$

  where:
  - $\mathbf{v_N}$ (the "center") is also called a routing object, and
  - $\mathbf{r_N}$ is called the (covering) radius of the region

- The set of indexed points p that are reachable from node N are guaranteed to have $d(p,v_N) \leq r_N$




- This immediately makes it possible to apply the pruning principle:

  **If $d(q,v_N) > r_N + r$ then prune node N**

# Entries of leaf and internal nodes

- Each node N stores a variable number of *entries*

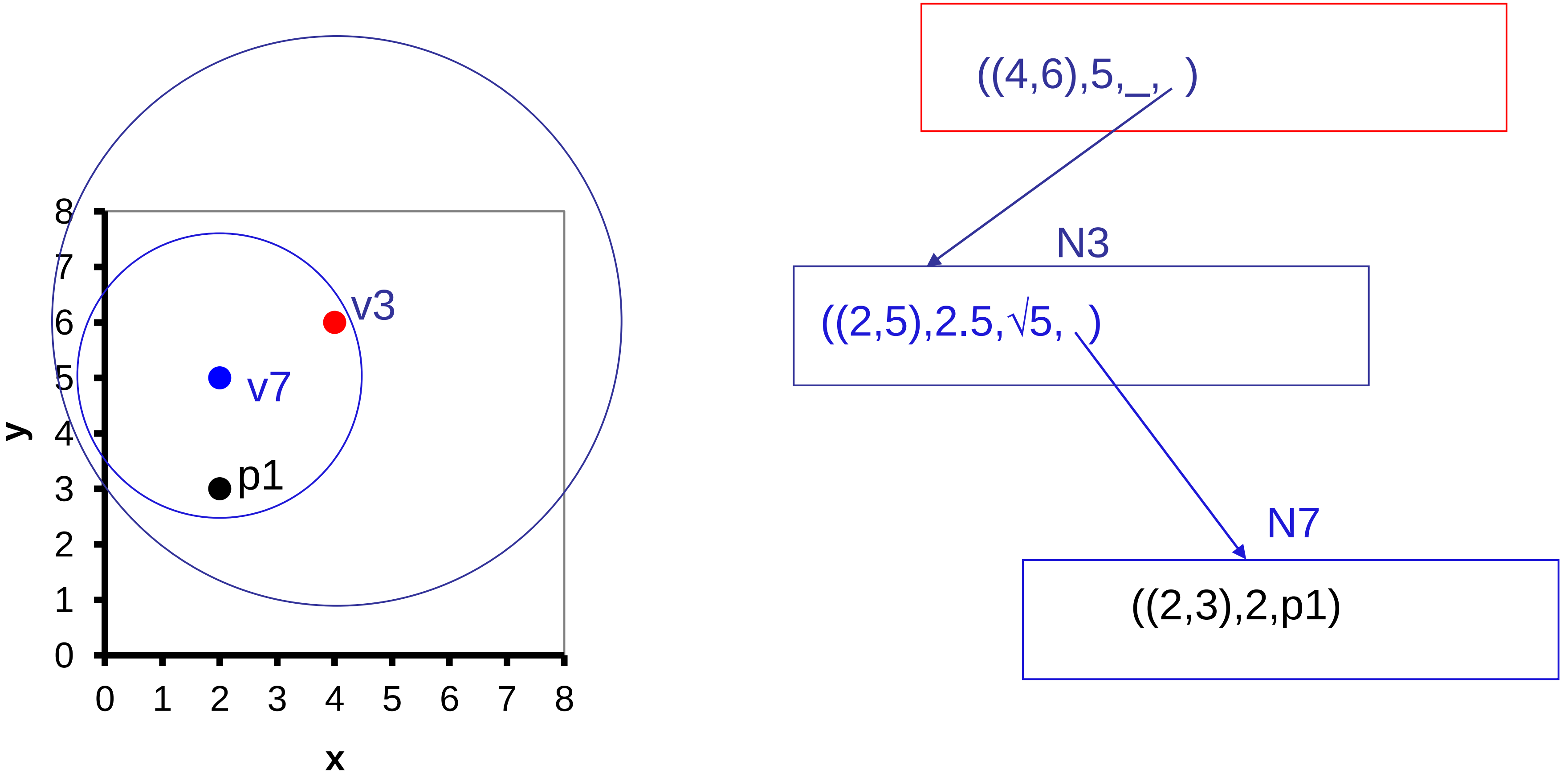
Leaf node:

- An entry E has the form E=(ObjFeatures,distP,TID), where
  - ObjFeatures are the feature values of the indexed object
  - distP is the distance between the object and its parent routing object (i.e, the routing object of node N)

Internal node:

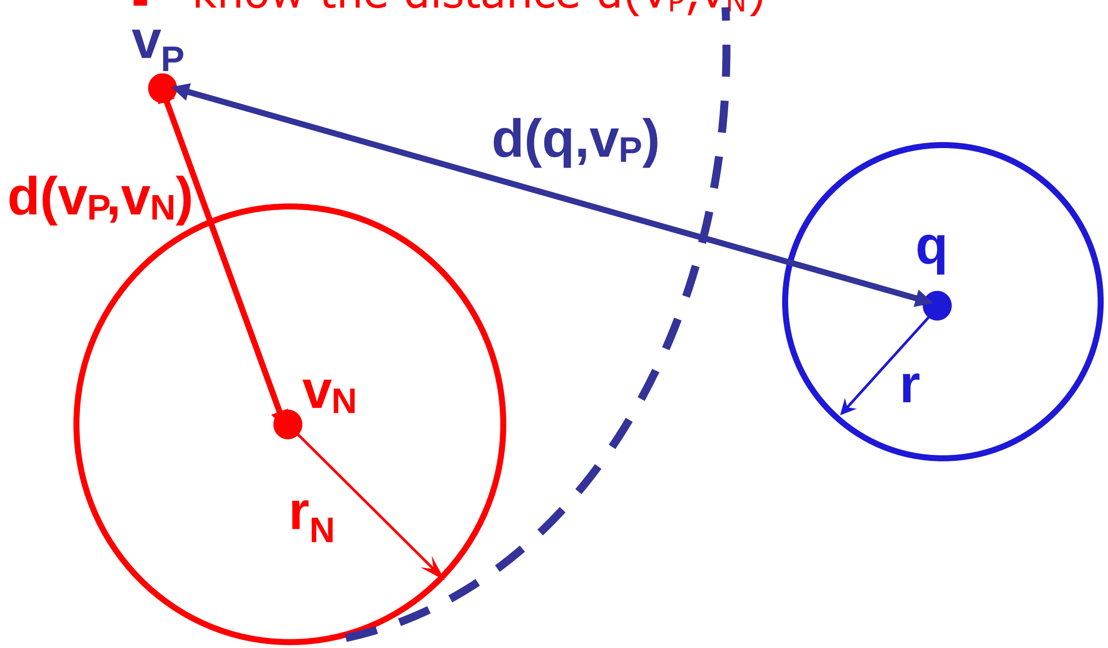
- E=(RoutingObjFeatures,CoveringRadius,distP,PID), where
  - RoutingObjFeatures are the feature values of the routing object
  - CoveringRadius is the radius of the region
  - distP is the distance between the routing object and its parent routing object (undefined for entries in the root node)

# Entries: an example



# Fast pruning based on distP

- Pre-computed distances distP are exploited during query execution to save distance computations
- Let $v_P$ be the parent (routing) object of $v_N$
- When we come to consider the entry of $v_N$, we
  - have already computed the distance $d(q,v_P)$ between the query and its parent
  - know the distance $d(v_P,v_N)$




From the triangle inequality it is:
$d(q,v_N) \geq |d(q,v_P) - d(v_P,v_N)|$

Thus we can prune node N ***without computing*** $d(q,v_N)$ if

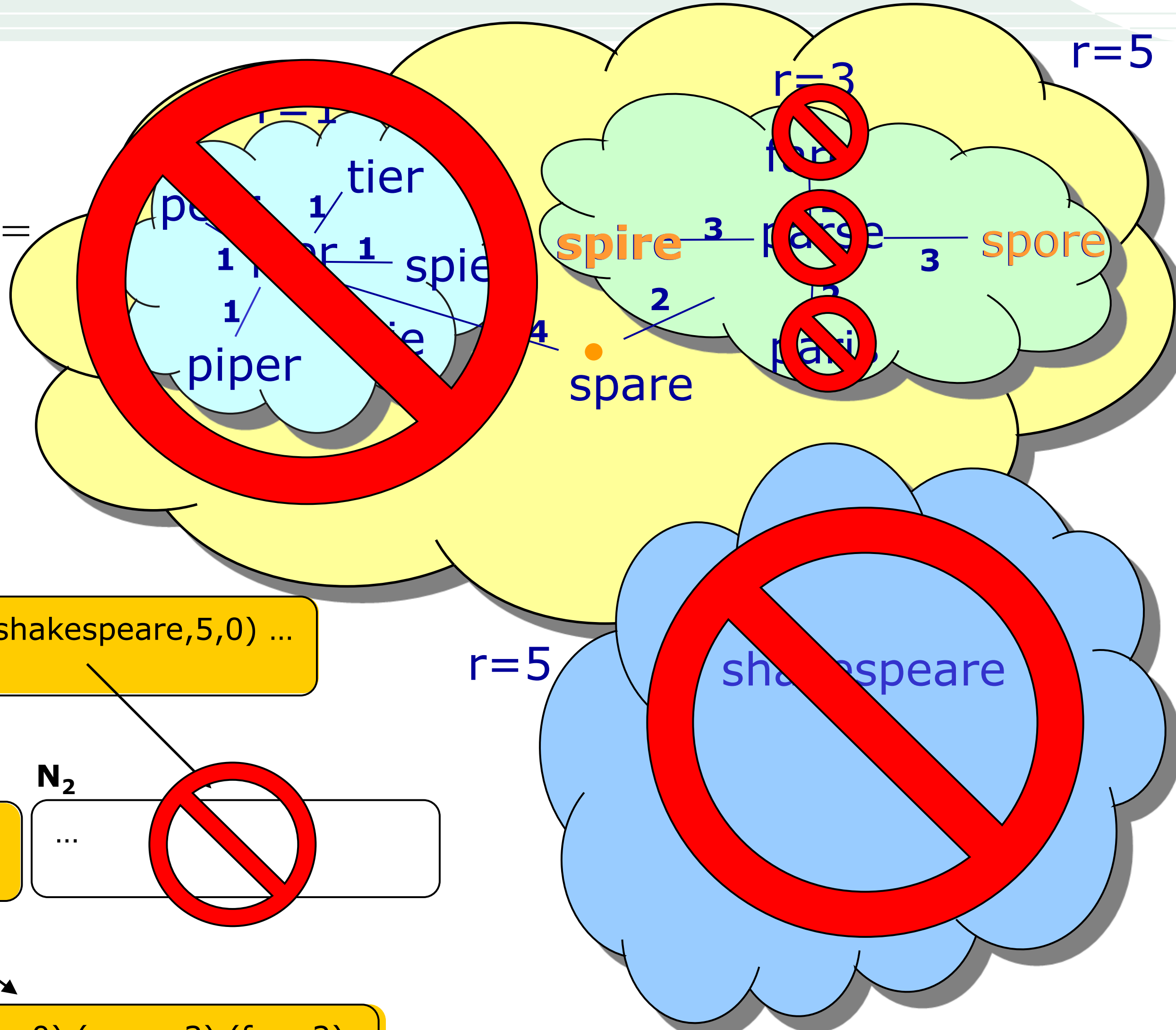
$$|d(q,v_P) - d(v_P,v_N)| > r_N + r$$

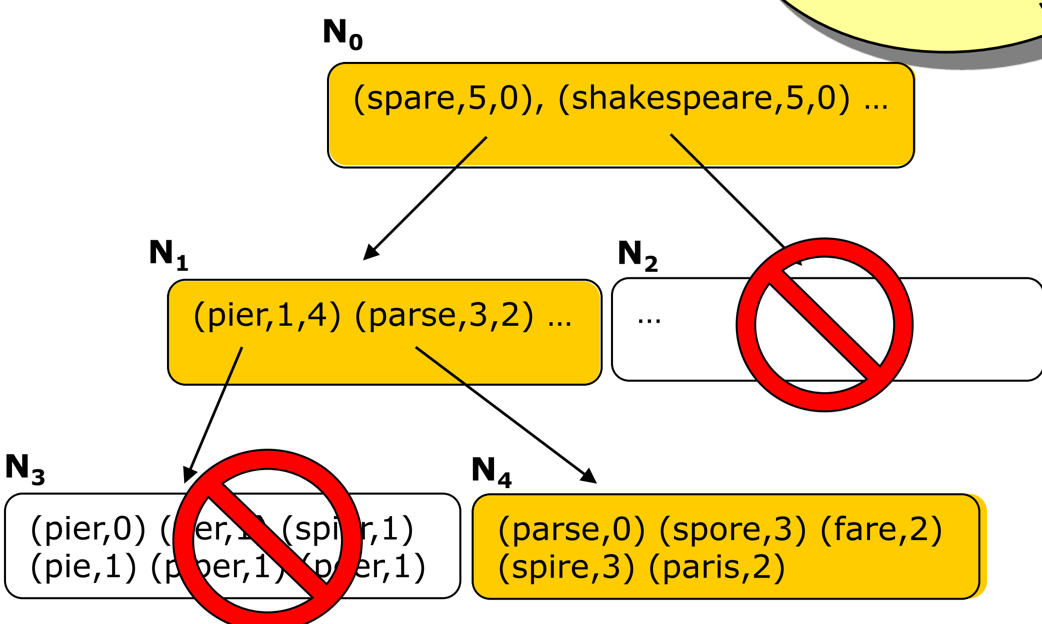
# Example (edit distance)

query = “spire”, r = 1

d(“spire”, “shakespeare”) =
= 7 ≤ 5 +1

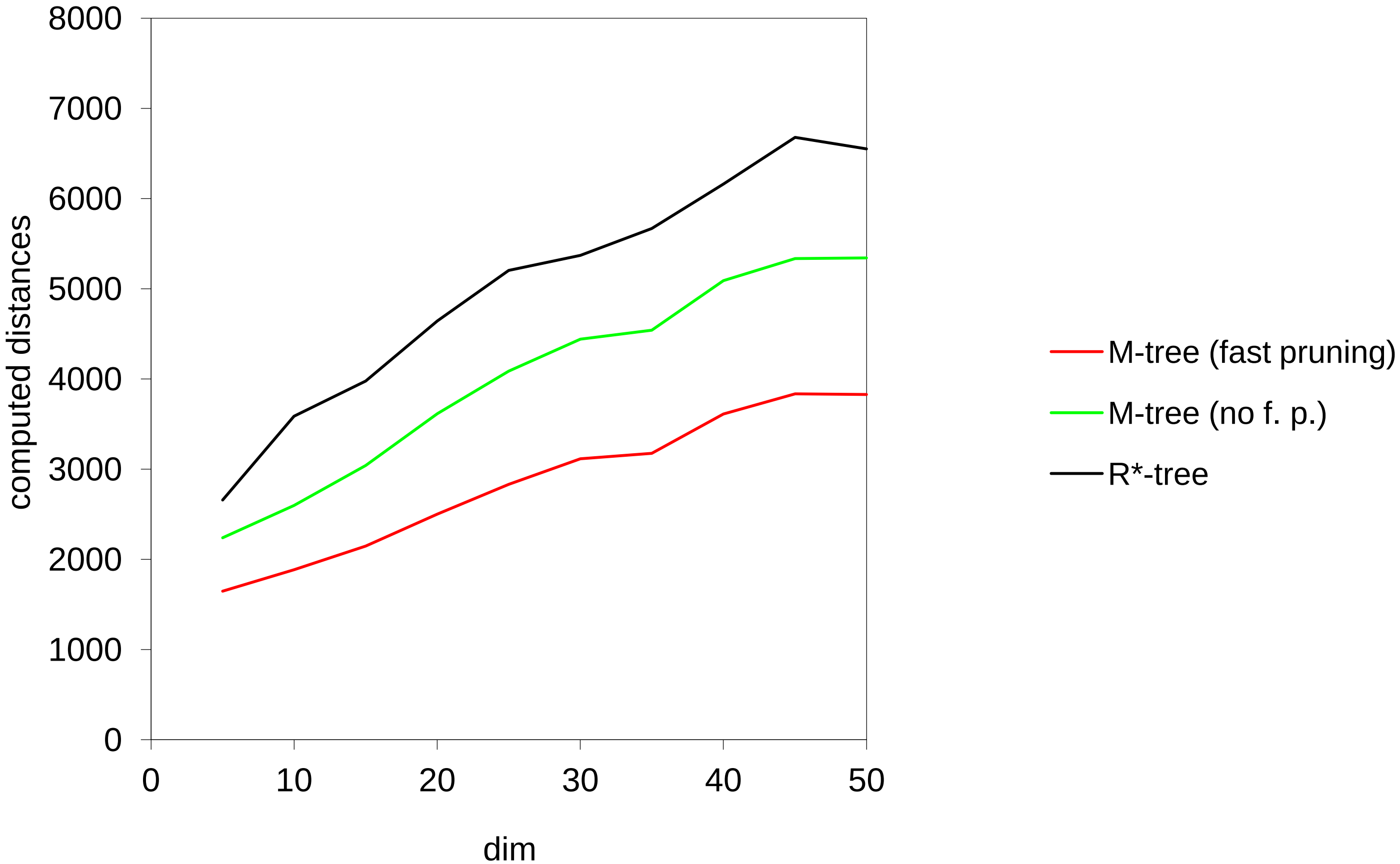
| d(“spire”, “spare”) –
d(“parse”, “spare”) | =
= | 3 – 0 | = 3 ≤ 3 +1



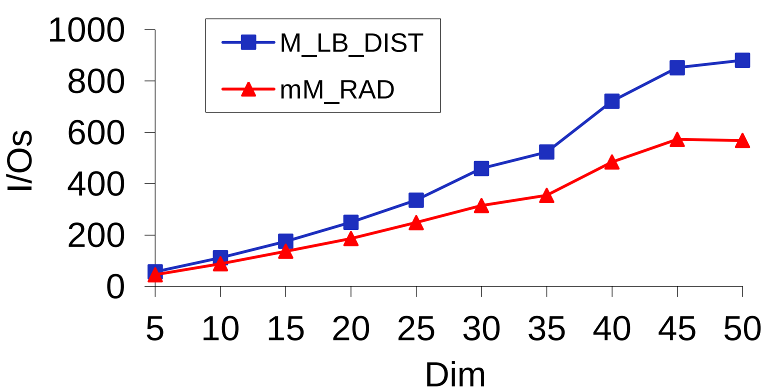
# Experimental results

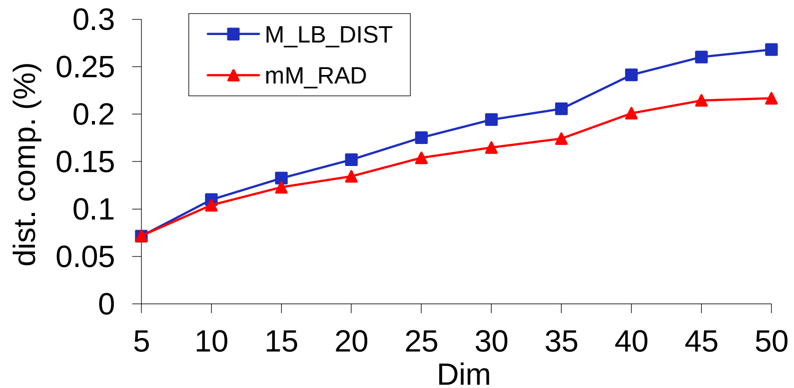
- Synthetic datasets (10 Gaussian clusters)
- Up to 40% cost reduction with fast pruning



# Insertion and split (sketch)

- The procedure to insert a new object is based on the ChooseSubtree method
- The Penalty method considers the increase of the covering radius needed to accommodate the new object
  - Remind: no “volume” can be computed!
- For managing a split, there are several alternatives, among which [CPZ97]:
  - mM_RAD minimize the maximum of the two resulting radii
  - M_LB_DIST choose the closest and the farthest object from $v_N$
- Experiments demonstrate that mM_RAD is the best

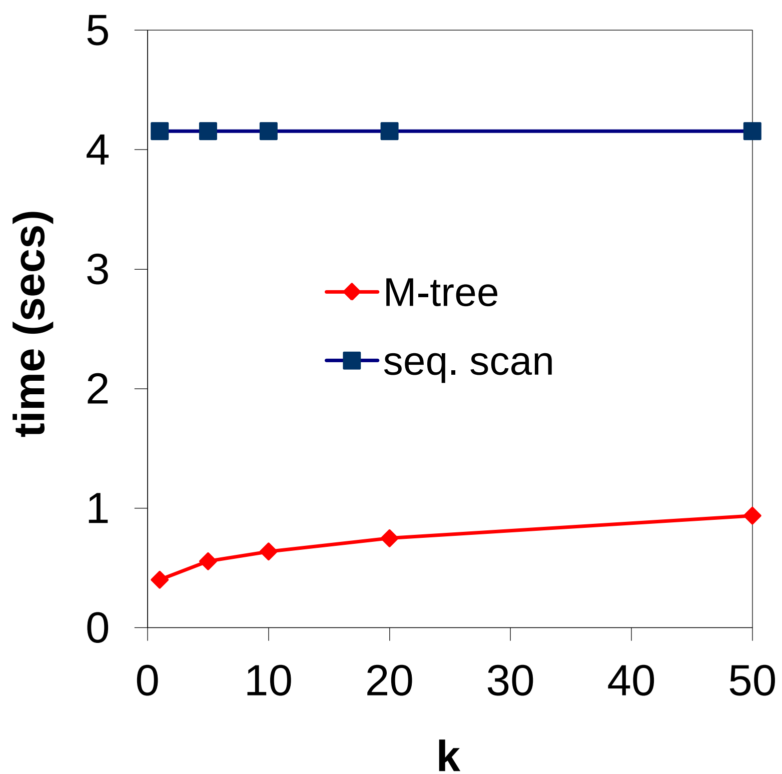


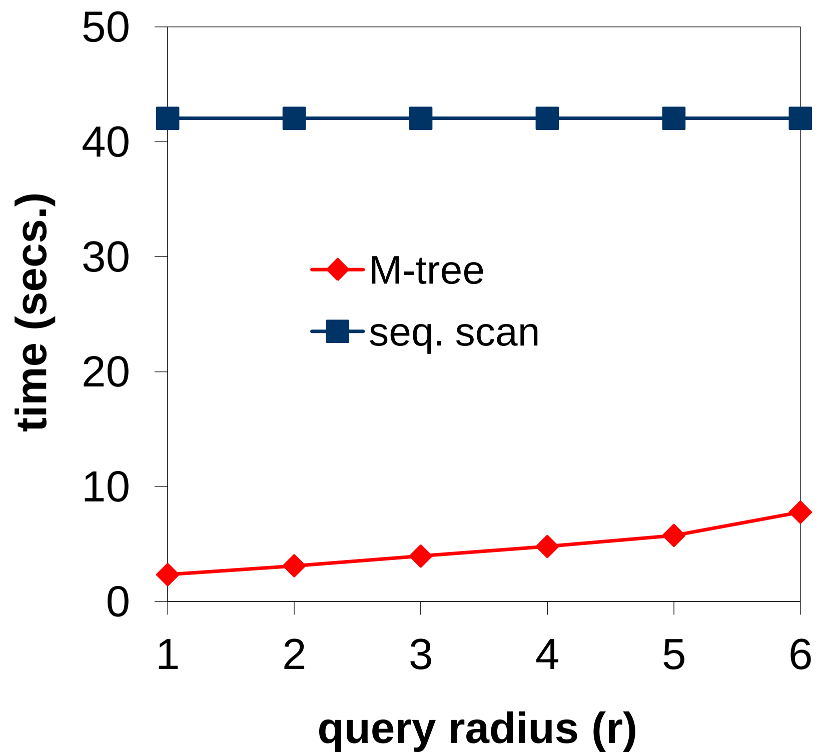



# Experiments (k-NN and range queries)

- 68,000 color images
- 32-dim (color histograms), $L_2$
- 161,212 text rows
- Edit distance

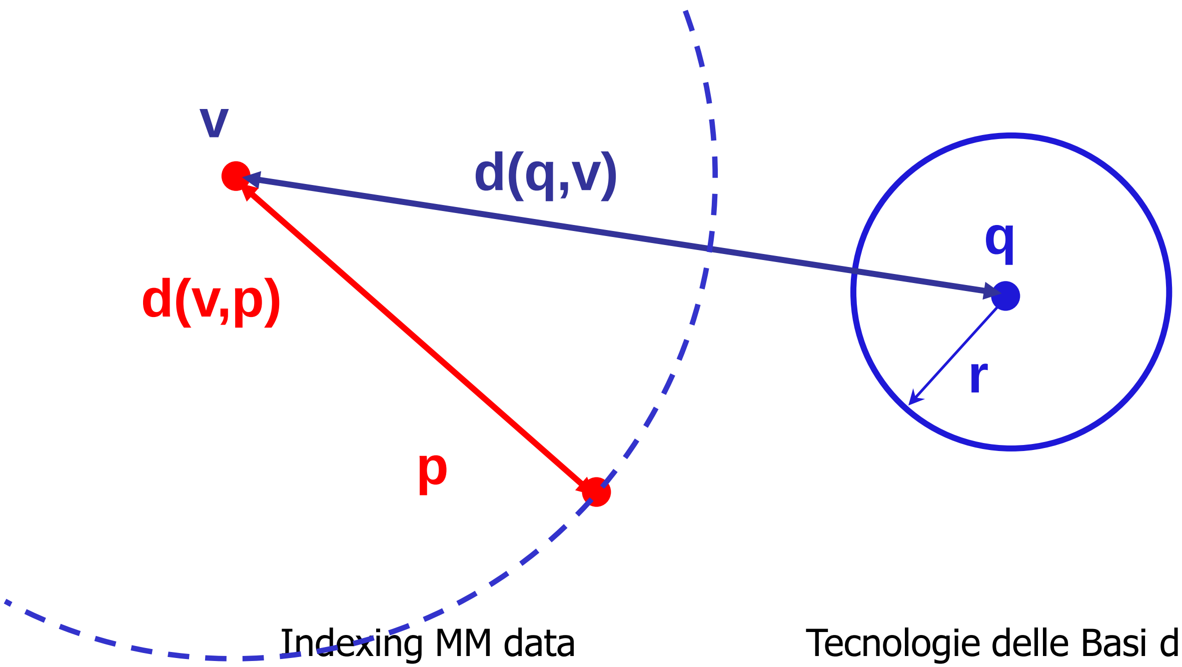
**The logic of search algorithms is the one already seen for range and k-NN queries with the R-tree**



# Pivot-based indexing (pivot tables)

- It is a sequential (main memory) structure
- It exploits the ball partitioning principle (without recursion)
- The idea:
  - choose a (pivot) point v
  - compute (and store) distances between all data points and v, d(v,p)
  - at query time, compute d(q,v)
  - if |d(q,v) – d(v,p)| > r then p cannot be part of the result



# Pivot tables (AESA/LAESA)

- The same principle can be applied to another pivot, and to another, and so on
- In AESA (*Approximating and Eliminating Search Algorithm*, Vidal 1984) every data point acts as a pivot
  - Pros: for each data point we have an increasing number of pivots (every time we cannot prune a data point p, we compute d(q,p), thus p can be used as a pivot for subsequent points)
  - Cons: the table grows quadratically with the data size
- In LAESA (*Linear AESA*, Micó and Oncina 1994) the pivots are a fixed number (*M*) of randomly chosen points (possibly not in the dataset)
  - A point p can be excluded if $|d(q,v_i) - d(v_i,p)| > r$ for any pivot $v_i$
  - This is equivalent to have a constraint $d(v_i,p) \in [d(q,v_i) - r, d(q,v_i) + r]$ for any pivot $v_i$
  - This is equivalent to a query window in a *M*-dimensional space
- The Omni-indices use "regular" structures (B-trees, R-trees, etc.) to index the *M*-dimensional space
- The choice of "good" pivots and of an optimal value for *M* are still open research issues

# High-dimensional spaces (1)

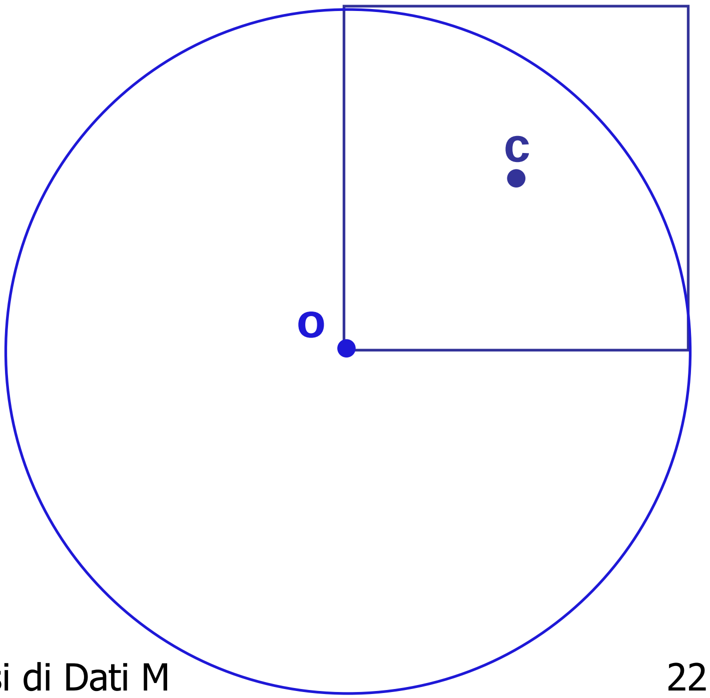
- The geometry of high-dimensional spaces is intriguing, since our common-sense intuitions fail, as the following examples show

1st example: "is the center in the sphere?"

- Consider the unitary hypercube $[0,1]^D$ with center c = (0.5,…,0.5), and the D-dimensional hypersphere $S^D$ centered in the origin o = (0,…,0) and with radius r = 1.
- Our intuition, and the figure as well, confirms that for D=2 c is inside $S^D$
- Let's see what happens when D grows:

$$L_2(c,o) = \sqrt{\sum_{i=1,D} 0.5^2} = \sqrt{D \times 0.5^2} = 0.5 \times \sqrt{D}$$

- Thus,
  **when D > 4 c is external to the sphere!**

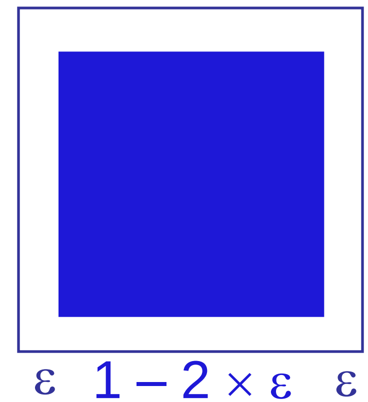


# High-dimensional spaces (2)

2nd example: "where are the points?"

- Consider again the unitary hypercube $[0,1]^D$
- Now, take a hypercube B of side $1 - 2 \times \varepsilon$ and center c = (0.5,…,0.5)
- The volume of B grows like

$$Vol(B) = (1 - 2 \times \varepsilon)^D$$




- As the table shows, even for (very) small ε values, Vol(B) sharply reduces

| ε \ D | 2 | 50 | 100 | 500 | 1000 |
|---|---|---|---|---|---|
| 0.1 | 0.64 | 1.43E-05 | 2.04E-10 | 3.51E-49 | 1.23E-97 |
| 0.05 | 0.81 | 0.01 | 2.66E-05 | 1.32E-23 | 1.75E-46 |
| 0.01 | 0.96 | 0.36 | 0.13 | 4.10E-05 | 1.68E-09 |

- If we have N points uniformly distributed over $[0,1]^D$, then only a fraction equal to Vol(B) will be contained, on the average, in B
- Thus, **all points are close to the surface of $[0,1]^D$ !**

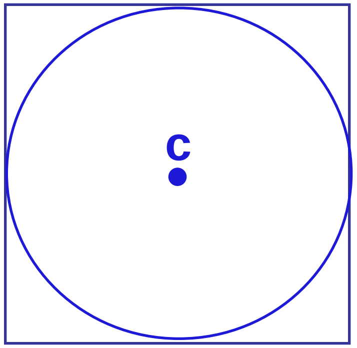
# High-dimensional spaces (3)

3rd example: "How big a sphere is?"

- Consider the unitary hypercube $[0,1]^D$ and the D-dimensional hypersphere $S^D$ centered in c = (0.5,…,0.5) and with radius r = 0.5
- The volume of $S^D$ can be computed as (D even):

$$Vol(S^D) = \frac{\pi^{D/2} \times 0.5^D}{(D/2)!}$$

- The following table (from [WSB98]) shows, for various values of D and assuming that points are uniformly distributed over $[0,1]^D$:
  - The volume of $S^D$, $Vol(S^D)$
  - The number of points N needed to have, on the average, **at least 1 point** in $S^D$ (this is just $1/Vol(S^D)$)



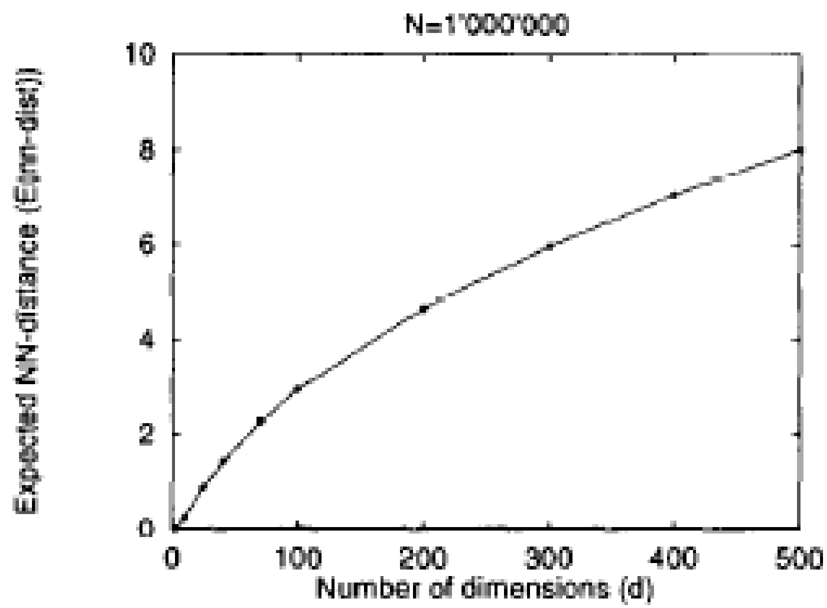

| D | Vol(S^D) | N |
|---|---|---|
| 2 | 0.785 | 1.27 |
| 4 | 0.308 | 3.24 |
| 10 | 0.002 | 401.50 |
| 20 | 2.46E-08 | 40631627 |
| 40 | 3.28E-21 | 3.05E+20 |
| 100 | 1.87E-70 | 5.35E+69 |

- Thus, **the number of points should grow exponentially to have at least 1 point in $S^d$!**

# High-dimensional spaces (4)

4th example: “How far is the nearest neighbor?”

- Continuing with the previous example, we can compute the expected (Euclidean) distance of the nearest neighbor of the center c=(0.5,…,0.5) of $S^D$
- The following graph (from [WSB98]) shows how the NN distance grows with D when N = $10^6$

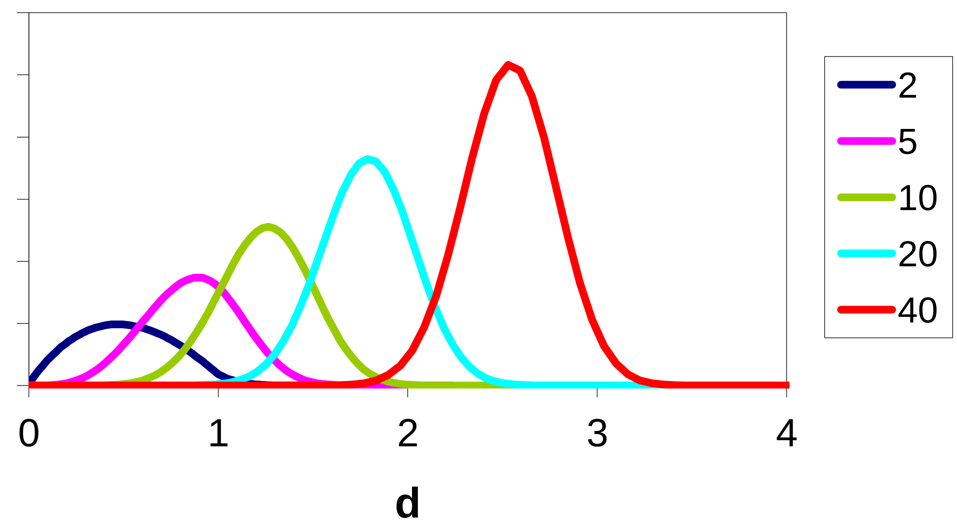



- Thus, **the closest point is far away!**

# High-dimensional spaces (5)

5th example: “How far are the other points?”

- We now plot the distance distribution of the dataset, for various values of D
- The distance distribution shows, for a given value of d, which is the percentage of points whose distance is d



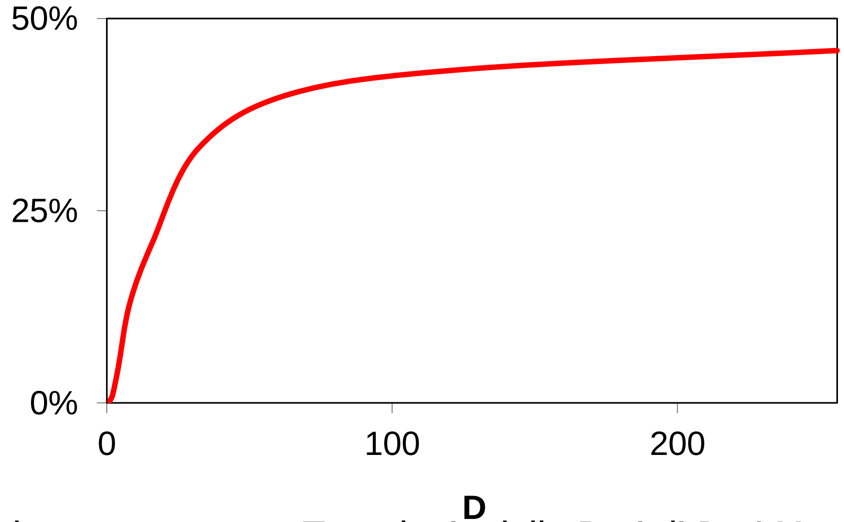

- It can be observed that when D grows, the variance of distances decreases
- Thus, **in high-dimensional spaces all points tend to have the same distance from the query!**

# Basic facts about high-dim. spaces (1)

- The analysis in [WSB98] demonstrates that, no matter how smart you are in designing a new index structure, there always exists a value of D such that the index performance will deteriorate, and sequential scan will become the best alternative!
- However, the analysis applies to uniformly distributed datasets and Euclidean distance…
- If data are not uniformly distributed (as it always happens!), then the authors argue that their analysis still applies, provided one considers the "intrinsic dimensionality" of the dataset
- The concept of "intrinsic dimensionality" is not precisely definable, intuitively it is the "true dimensionality" of our data
  - E.g.: a line has intrinsic dimensionality 1, regardless of D
- Some attempts to characterize the intrinsic dimensionality of a dataset have been based on the concept of fractals (e.g., see [FK94])

# Basic facts about high-dim. spaces (2)

- From a more pragmatical point of view, experimental results obtained with both spatial and metric indexes confirm that high-dimensional datasets are often a nightmare!
- This is the so-called “**dimensionality curse**”!
- For the structures we have seen (R-tree and M-tree), what is observed is an **incredible amount of overlap between the regions of index nodes**
  - The graph shows the percentage of M-tree regions that enclose a query point q, i.e., those regions for which $d_{MIN}(q,Reg(N)) = 0$
  - Thus, **all such regions can never be pruned during a k-NN search!**



# Partitioning without overlap

- If we partition the $[0,1]^D$ space into non-overlapping regions, similar problems arise
- For instance, consider a uniform distribution of points, and assume we split a dimension in the mid-point 0.5 (thus, each time we double the number of regions). We can split at most $D' = \lceil \log_2 N \rceil$ dimensions
- Consider the region: $Reg = [0,0.5] \times \ldots \times [0,0.5] \times [0,1] \times \ldots \times [0,1]$

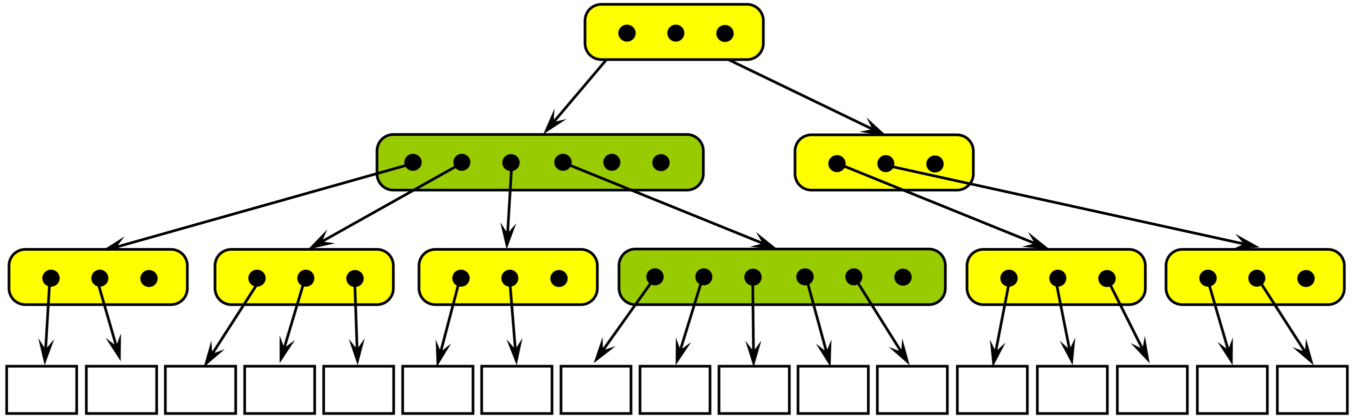
  whose farthest point is $q = (1,\ldots,1)$
- The Euclidean distance of q from Reg is:

$$L_2(Reg, q) = \sqrt{\sum_{i=1,D'} (1-0.5)^2} = \sqrt{D' \times 0.5^2} = 0.5 \times \sqrt{D'} = 0.5 \times \sqrt{\lceil \log_2 N \rceil}$$

- With $N = 10^6$ we have $D'=20$ and $L_2(Reg,q)=2.236$
- Since this is independent of D, whereas the expected NN distance grows with D, for values of D large enough ($D \geq 80$) Reg will be accessed, and this holds for any other region!

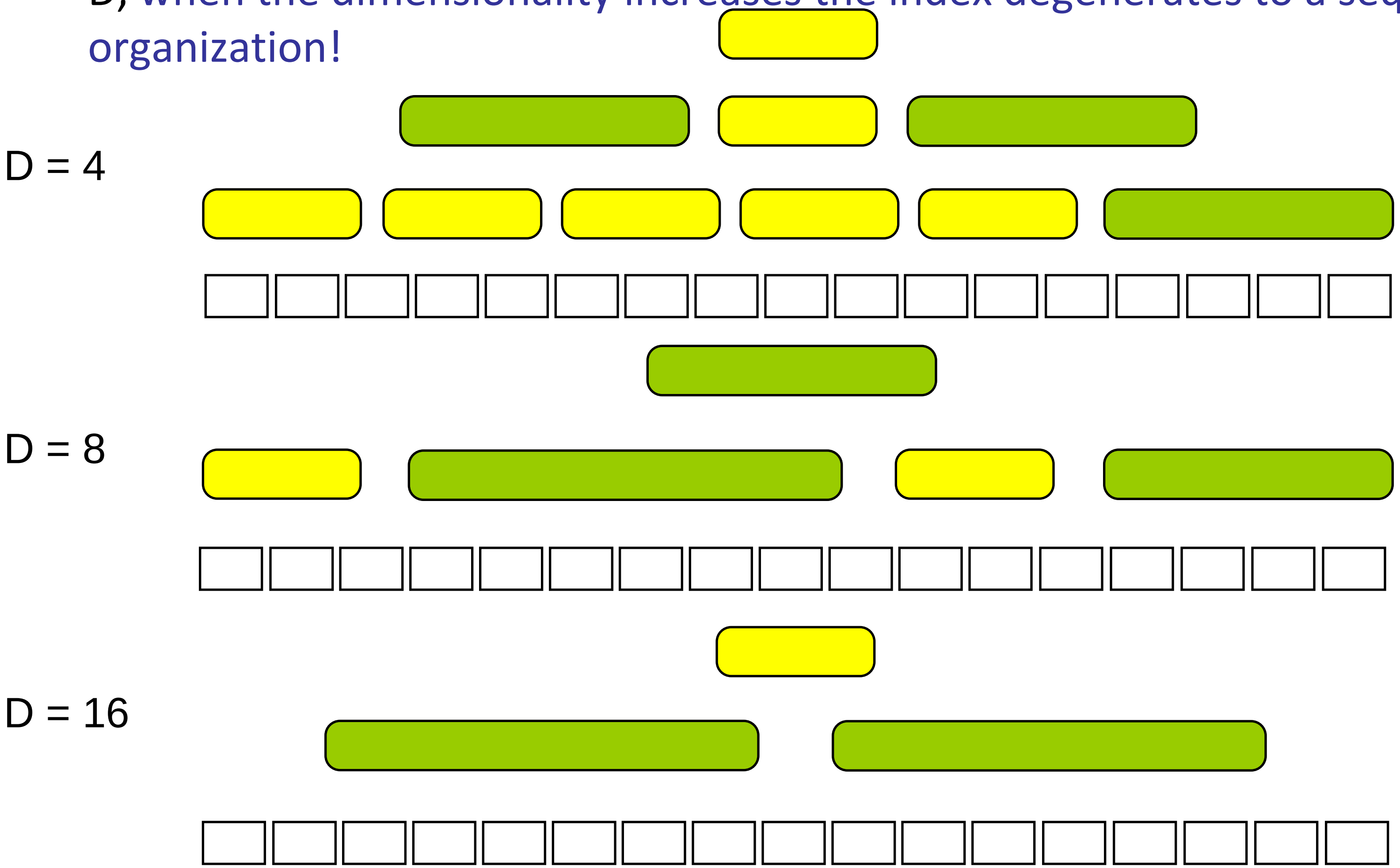
# The X-tree [BKK96]: basic idea

- The X-tree is an evolution of the R-tree, aiming to deal with the "overlap problem"
- When a node has to be split, if an overlap-free split is possible then it is performed as usual, otherwise a new, larger, *super-node*, is allocated
  - Thus, now we have nodes of variable size
- The price to be paid is that searching within a super-node is more costly than searching within nodes



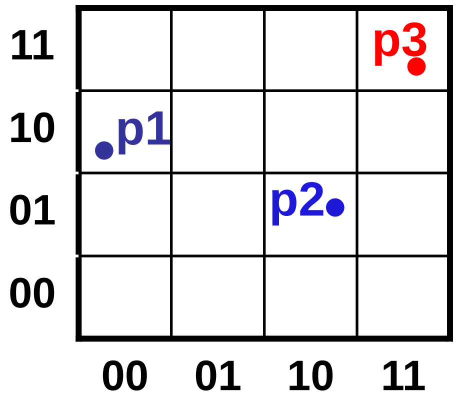
# The X-tree: what happens when D grows

- Although the X-tree performs better than the R-tree for medium values of D, when the dimensionality increases the index degenerates to a sequential organization!



# The VA-file (Weber, Schek & Blott, 1998)

- The basic idea of the VA-file [WSB98] is to speed-up the sequential scan by exploiting a “Vector Approximation”
- Each dimension of the data space is partitioned into $2^{b_i}$ intervals using $b_i$ bits
  - E.g.: the 1st coordinate uses 2 bits, which leads to the intervals 00,01,10, and 11
- Thus, each coordinate of a point (vector) requires now $b_i$ bits instead of 32
- The VA-file stores, for each point of the dataset, its approximation, which is a vector of $\sum_{i=1,D} b_i$ bits




**Data space**

**Feature values**

| | |
|---|---|
| p1 | 0.1 0.6 |
| p2 | 0.7 0.4 |
| p3 | 0.9 0.3 |

**VA-file**

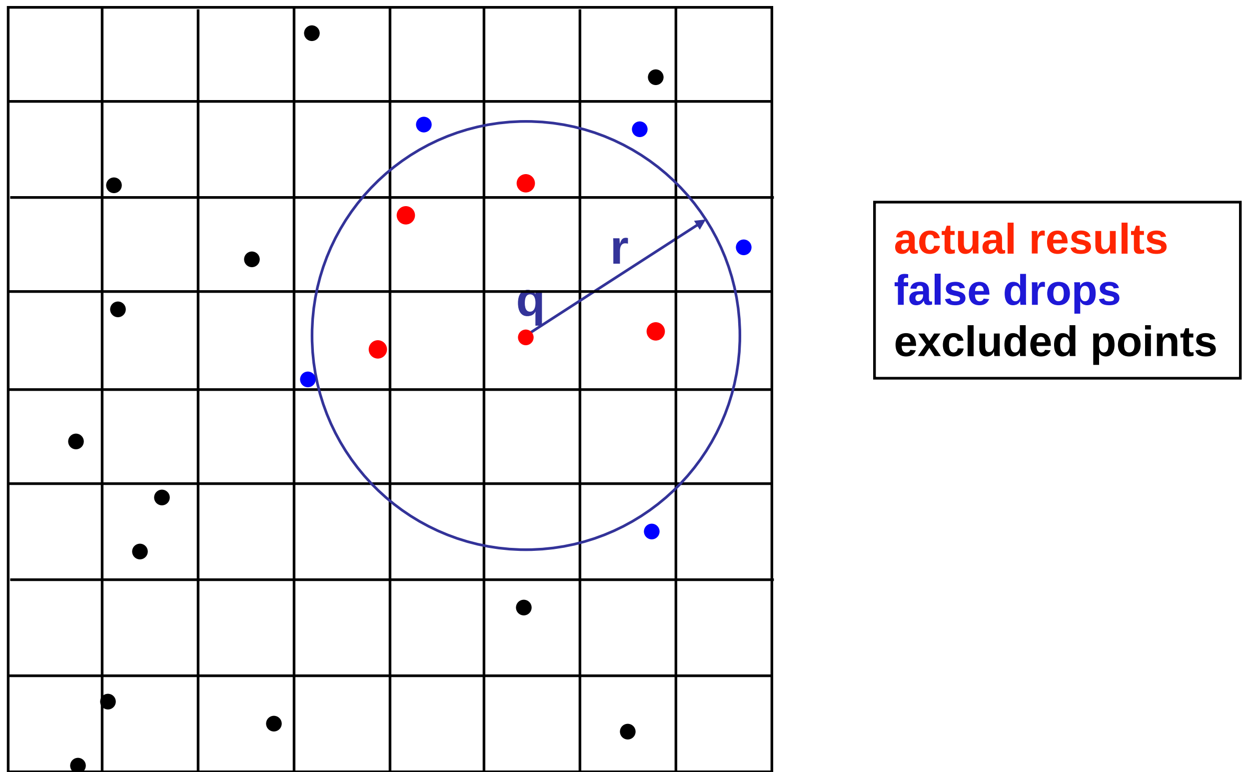
| | |
|---|---|
| p1 | 00 10 |
| p2 | 10 01 |
| p3 | 11 11 |

# The VA-file: query processing

- Query processing with the VA-file is based on a filter & refine approach
- For simplicity, consider a range query

Filter: the VA file is accessed and only the points in the regions that intersect the query region are kept

Refine: the feature vectors are retrieved and an exact check is made



# Conclusions (?)

- The issue of efficiently indexing complex datasets is far from having been solved
- Starting from the end of 90's, many solutions have been proposed, and new ideas have emerged
- Unfortunately, the absence of a well-defined and accepted benchmark makes it almost impossible to compare all such solutions

- The basic lesson to be learned is that, no matter how a structure has been cleverly designed, ultimately it has to be contrasted with the sequential scan!
- Thus, be skeptical if someone claims to have designed an index showing "superior performance" w.r.t. the others: always look if sequential scan has been taken as a competitor!