UNIVERSITÄT BASEL

Computer Science / 15731-01 / 2019

Multimedia Retrieval

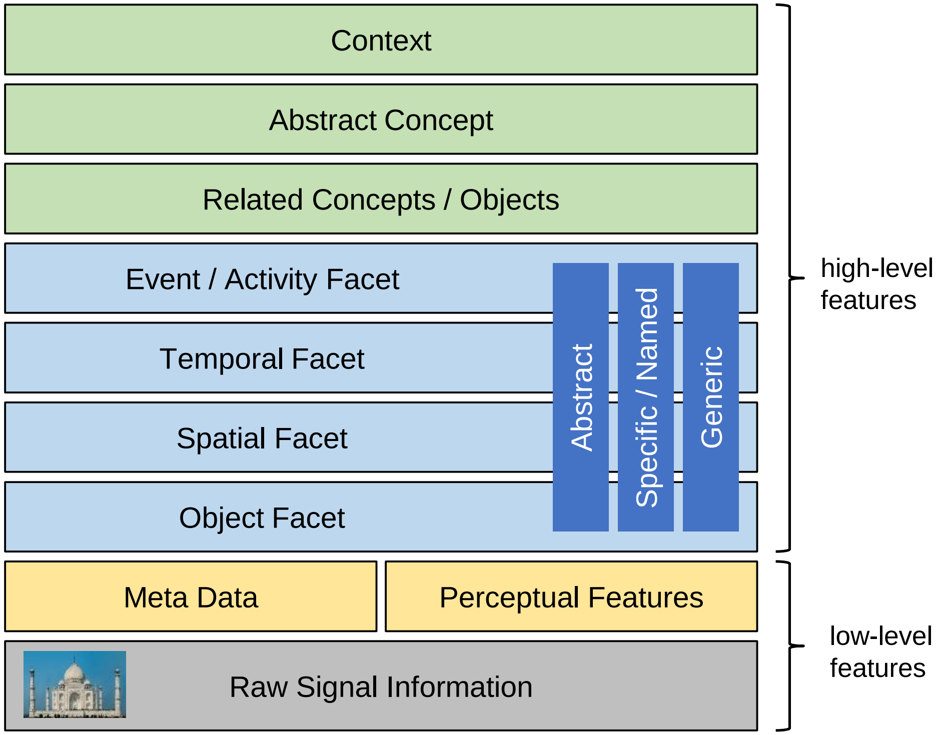
Chapter 6: Similarity Search

Dr. Roger Weber, roger.weber@ubs.com

- 6.1 Overview
- 6.2 High-dimensional Feature Search
- 6.3 Curse of High Dimensionality
- 6.4 Complex Similarity Search
- 6.5 References

6.1 Overview

- In the previous chapters, we have discussed various methods to extract features and to retrieve relevant documents. We have distinguished low-level features (close to signal information) and high-level features (close to perceptual and context related interpretation of the user).
- The level system on the right side is valid for all media types including text, image, audio, and video. We have seen so far:
  - Low-level features for text with set-of-word and bag-of-word description
  - Low-level features for images, audio files, and videos with multi/high dimensional data summarizing perceptual aspect of the raw signal information like color moments, spectral bandwidth, or optical flow
  - High-level features that extract terms, assign class memberships, or cluster data
  - Various methods to retrieve text related queries and method to use link information to rank documents



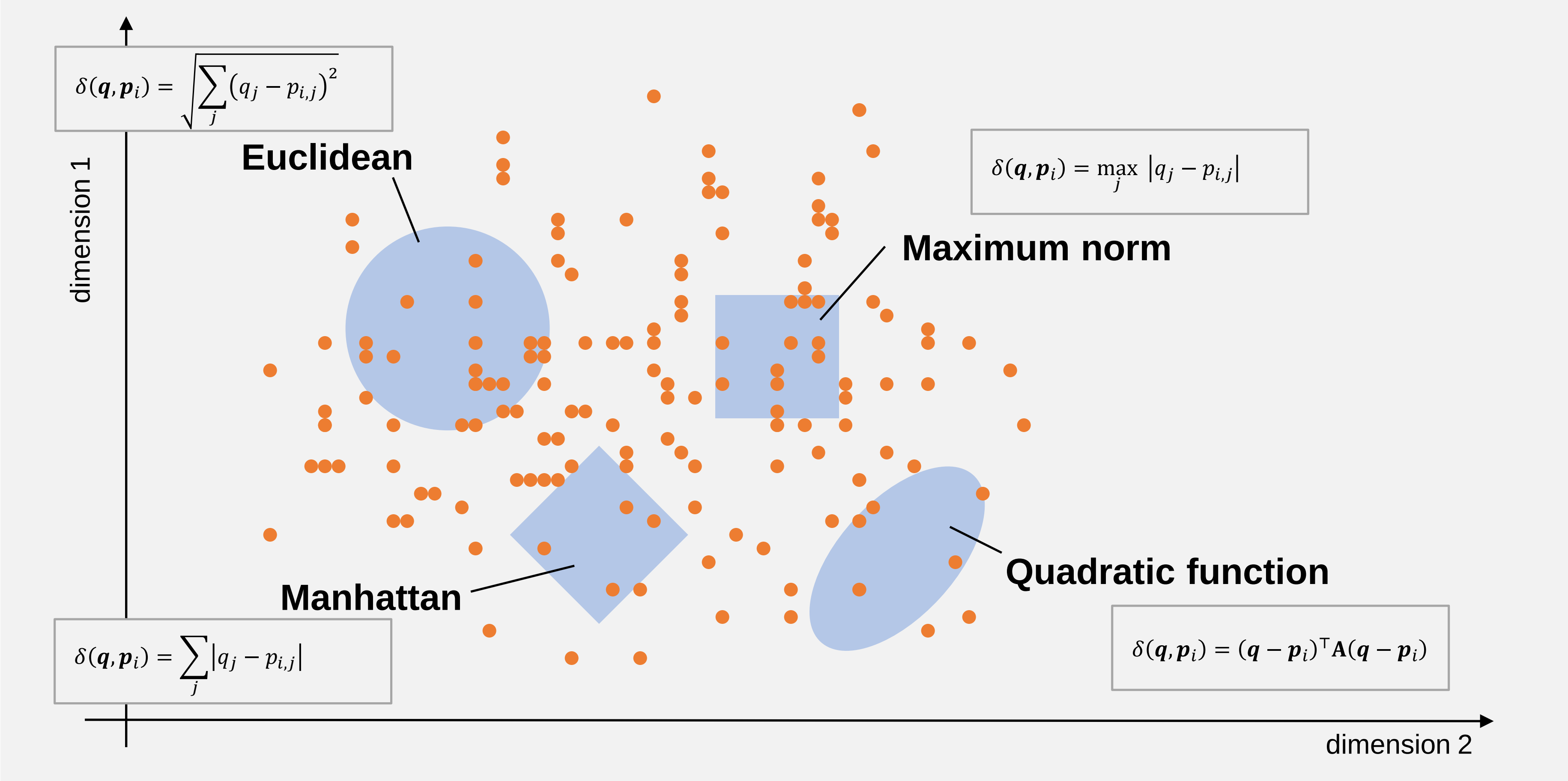
- But we have not yet discussed how to
  - Search for high-level features
  - Search for multi/high dimensional features
  - Search for documents/objects if several features (and objects) are given

- High-level features are often straightforward:
  - Classification methods assign a set of class memberships (with probabilities) to an object. These memberships can be used either as attributes (with predicates to search for) or as new terms which provide further annotations next to the descriptions being given
  - Clustering assigns objects to groups (or several groups with probabilities). Cluster membership can be used to reduce the number of objects during retrieval (only look in the same cluster). In contrast to classification, the semantics of clusters are unknown (or may not even exist)
  - Neural network can return predicate values (face? yes or no) or class memberships (one class or membership with probabilities). In both cases, we can transform the output into either an attribute value (with predicates to search for) or extract new terms (with probabilities interpreted as occurrences)
- In other words, we can map many of the high-level features back into a “term” domain and use any of the known (text) retrieval methods to search for relevant objects. A term like “cat” may match to the result of a neural network that can detect cats in images. A term like “jazz” can match the classification of a decision tree that categorizes music plays.
  - For term based queries, this closes the semantic gap between the user and raw signal information. How would you otherwise define what “jazz” means in the time or frequency domain of an audio signal?
  - If reference objects are used (e.g., ‘search for more like this’), the high-level features provide a set of keywords to look for. If several objects are given, occurrences of features work similar to term occurrences within queries for traditional retrieval models
- On the other side, if we map high-level features to attributes, queries define predicates on these attributes. For instance, *hasFace* captures whether a face is visible in an image or not. Any database provides capabilities to search for objects that fulfill several predicates on their attributes.

- Things become more difficult, however, if low-level features are taken into account or if different feature types are combined. Examples:
  - I want to find black horses
  - I want a pop-song with even emphasis on bass and high pitches
  - I look for videos with fast red cars
  - I look for an action scene of my favorite actress
  - I look for slow rock songs from my favorite band
- In the following, we provide methods to search through multi/high dimensional feature data and to combine different type of features to express a complex search. We mostly assume the presence of a reference object (or several objects) and search for similar objects. In the most complex case, this may include predicates on attributes and key-words for text based search

6.2 High-dimensional Feature Search

- Many low-level features extract multi or high-dimensional feature vectors. In the following, we consider how to search for similar objects given only these low-level features.
- We already discussed the use of distance measures to define what similarity means. The figure below visualize this definition with different distance measures. Often the definition of the low-level feature includes the “right” distance measure to use. But we can select arbitrary measures that best match our information need.



$\delta(\boldsymbol{q},\boldsymbol{p}_i) = \sqrt{\sum_j (q_j - p_{i,j})^2}$
Euclidean
$\delta(\boldsymbol{q},\boldsymbol{p}_i) = \max_j |q_j - p_{i,j}|$
Maximum norm
Quadratic function
$\delta(\boldsymbol{q},\boldsymbol{p}_i) = (\boldsymbol{q}-\boldsymbol{p}_i)^\top \mathbf{A}(\boldsymbol{q}-\boldsymbol{p}_i)$
Manhattan
$\delta(\boldsymbol{q},\boldsymbol{p}_i) = \sum_j |q_j - p_{i,j}|$
dimension 1
dimension 2

- **Normalization**
  - Many low-level features compose different aspects (like moments, covariances) or are the combination of different low-level features (e.g. spectral bandwidth and spectral flux). In such cases, it is required to normalize the value ranges of each dimension to avoid that a single dimension (=aspect) dominates the similarity definition. Example:
    - In dimension d_1, all values are between 0 and 1.
    - In dimension d_2, all values are between 100 and 200.
    - If we do not normalize the two dimensions, then differences in d_2 will always dominate the distance measure. It is like d_1 (=other aspect) has no influence at all on what is similar.
  - Gaussian normalization is a simple method to achieve satisfactory results. To this end, we need to compute the mean value and variance of values along each dimension. We do not require the “correct” values, hence, it is sufficient to sample a large enough data set and then keep the values constant. It is also not difficult to use incremental methods to adjust mean values and variances whenever the data set changes.
  - The transformation from the original vector $\boldsymbol{p}_i$ to its normalized version $\widehat{\boldsymbol{p}}_i$ is as follows:

$$\widehat{\boldsymbol{p}}_{i,j} = \frac{\boldsymbol{p}_{i,j} - \mu_j}{\sigma_j}$$

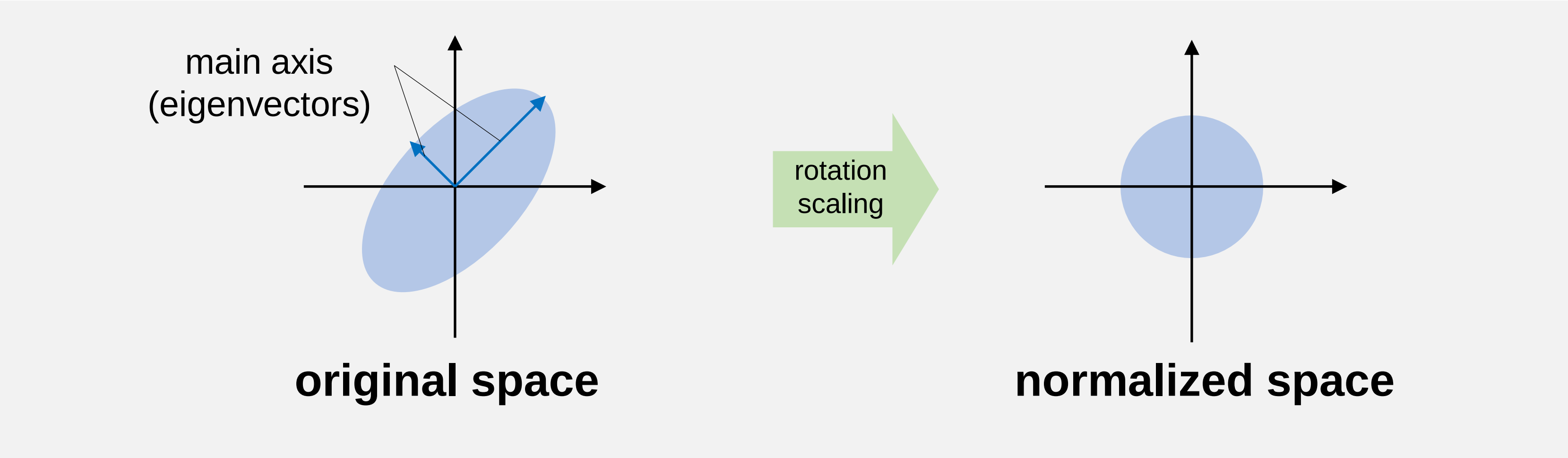
with μ_j and σ_j being the mean value and variance in dimension j, respectively. As we do the same for the vector $\boldsymbol{q}$ of the reference object and distances are based on differences in each dimension, the mean values drop out of the formula and the variances remain as scaling factor.

$$\delta(\boldsymbol{q}, \boldsymbol{p}_i) = \sum_j w_j \cdot \left| q_j - p_{i,j} \right| \qquad \text{with} \qquad w_j = \frac{1}{\sigma_j}$$

- Alternatively, we can normalize with the differences of the extreme values for each dimension:

$$w_j = \frac{1}{\max_i p_{i,j} - \min_i p_{i,j}}$$

- Quadratic distance functions are expensive to calculate and, as we will see, difficult to build an index for them.
  - Note that the matrix A in the quadratic function must be positive semi-definite, i.e., $\boldsymbol{x}^\top \mathbf{A}\boldsymbol{x} \geq 0$, to be considered as a useful distance measure. From linear algebra, we know that such matrices define a hyper ellipse and have real eigenvalues. Its eigenvectors define the main axis in the original space which then can be rotated and scaled to obtain a normalized space. Due to the properties of the Eigenvalue decomposition, the original quadratic function becomes a (squared) Euclidean distance measure.
  - Hence, after extraction of the low-level features, a simple rotation and scaling yields normalized features. We only keep these feature vectors so that we can apply a more efficient comparison.



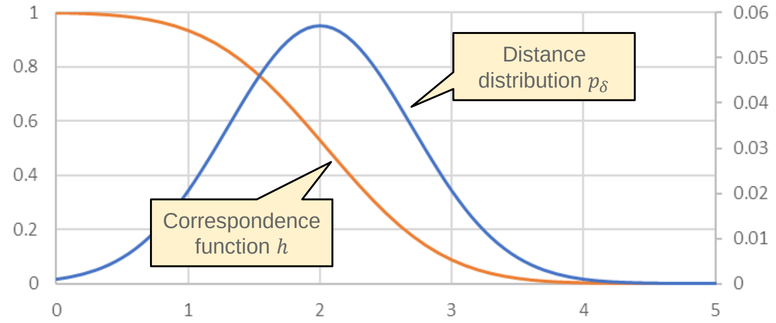
- If we want to obtain similarity values from the distances, we need a so-called **correspondence function** h**.** Let $\sigma(\boldsymbol{q},\boldsymbol{p}_i)$ denote a similarity function between query vector $\boldsymbol{q}$ and a media vector $\boldsymbol{p}_i$. The following properties must hold:
    - $\sigma(\boldsymbol{q},\boldsymbol{p}_i)$ is in the range $[0,1]$
    - $\sigma(\boldsymbol{q},\boldsymbol{p}_i)=0$ denotes total dissimilarity between query vector $\boldsymbol{q}$ and a media vector $\boldsymbol{p}_i$
    - $\sigma(\boldsymbol{q},\boldsymbol{p}_i)=1$ denotes maximum similarity between query vector $\boldsymbol{q}$ and a media vector $\boldsymbol{p}_i$
  - The correspondence function translates between distances and similarity values as follows

    $$\sigma(\boldsymbol{q},\boldsymbol{p}_i)=h(\delta(\boldsymbol{q},\boldsymbol{p}_i)) \qquad \delta(\boldsymbol{q},\boldsymbol{p}_i)=h^{-1}(\sigma(\boldsymbol{q},\boldsymbol{p}_i))$$

    It must fulfil the following constraints

    - $h(0)=1$
    - $h(\infty)=0$
    - $h'(x)\leq 0$ (h must be a decreasing function)
  - The best method to build a correspondence function is to use the distance distribution p_δ. We obtain the mapping by integrating the distribution function up to the given distance and subtract that value from 1. This guarantees that all constraints hold true:

$$h(x)=1-\int_0^x p_\delta(x)dx$$



Distance distribution p_δ

Correspondence function h

6.2.1 Nearest Neighbor Search Problem

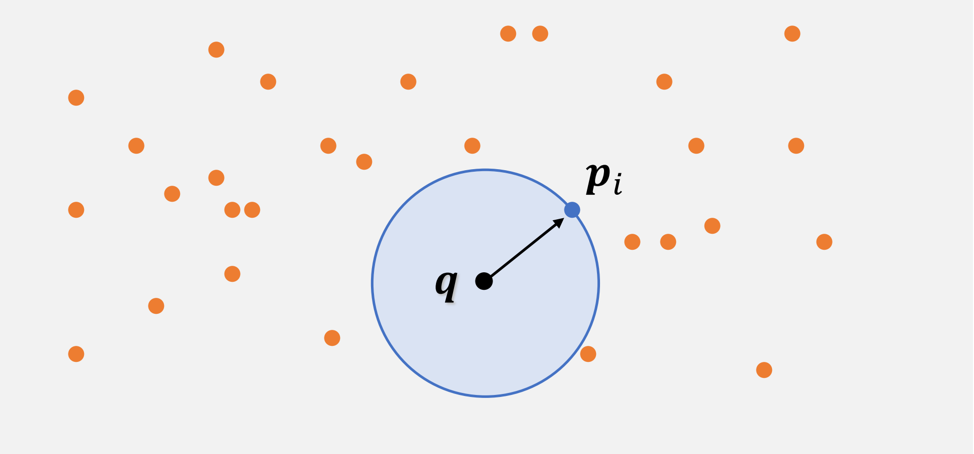
- Searching for the most similar object translates to a search for the object with the smallest distance, the so-called **nearest neighbor**. We note the reversed relationship between similarity values and distances:
  - large distances correspond to low similarity values
  - small distances correspond to high similarity values

  We can express similarity search as a nearest neighbor search:

Nearest Neighbor Problem:

- Given a vector $\boldsymbol{q}$ and a set $\mathbb{P}$ of vectors $\boldsymbol{p}_i$ and a distance function $\delta(\boldsymbol{q}, \boldsymbol{p}_i)$
- Find $\boldsymbol{p}_i \in \mathbb{P}$ such that:

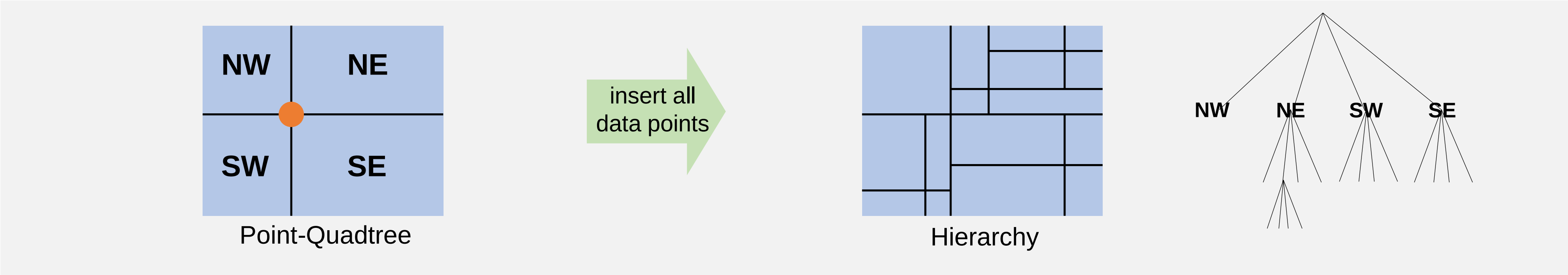
$$\forall j, \boldsymbol{p}_j \in \mathbb{P}: \delta(\boldsymbol{q}, \boldsymbol{p}_i) \leq \delta(\boldsymbol{q}, \boldsymbol{p}_j)$$



- In the following, we consider a small number of index methods that were proposed to index multi- and high-dimensional features and accelerate nearest neighbor search.
- We further discuss a generic algorithm that can find the nearest neighbor in optimal time and space given an index method and a distance measure.

6.2.2 Quadtree

- Finkel and Bentley described the Quadtree in 1974. Their goal was to develop an index structure in main memory capable of storing and searching for 2-dimensional data points. As a consequence, they have not considered an external storage format. Later on, the 2-dimensional Quadtree was extended to more dimensions but did not succeed.
- Structure:
  - The two dimensional space is divided with two orthogonal lines into four areas in the north-east (NE), north-west (NW), south-east (SE), and south-west (SW). The common edge of the four areas is the split point (intersection of the orthogonal lines). There were two method to define a split center: a) use a newly inserted data point to split the regions, or b) split the region in the center. The former had the advantage that the areas would naturally adapt to the distribution of the data points in the space. The latter was more easy to implement.
  - With each newly inserted data point, the Quadtree splits the region into four areas. After inserting all data points, a hierarchical structure divides the space into ever smaller areas. The resulting tree is not necessarily balanced but finding points is greatly accelerated over a brute-force linear scan. It is possible to bulk load data points in such a way that a perfectly balanced Quadtree results.

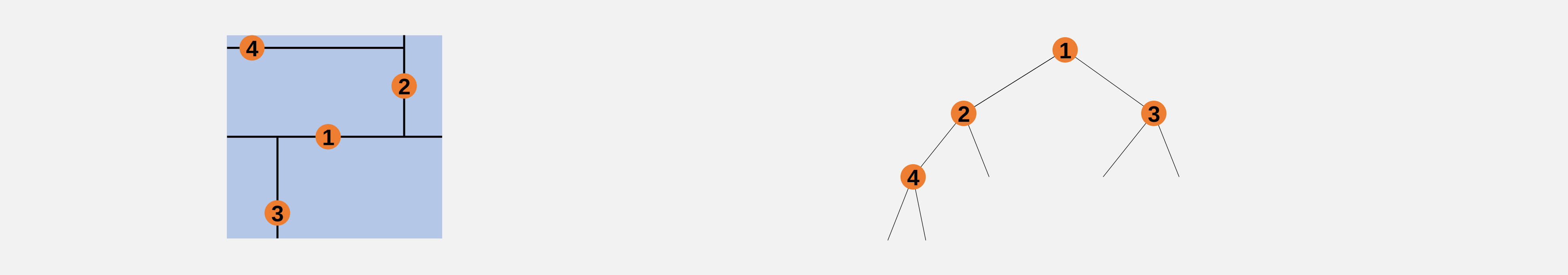


Point-Quadtree

Hierarchy

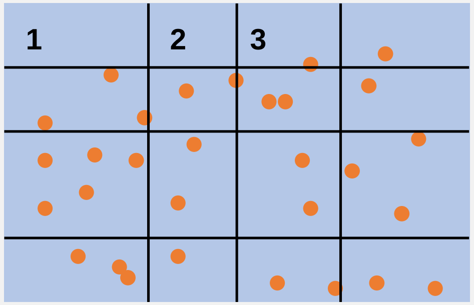
6.2.3 K-d-tree

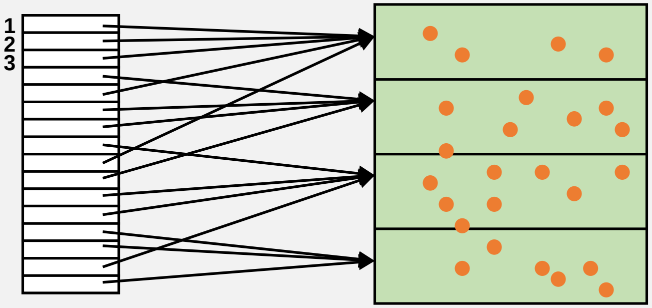
- Bentley then developed the k-d-tree in 1975 refining the original idea of the Quadtree to index higher dimensional spaces. Its extension, the k-d-B-tree was very popular given its additional balanced tree characteristics and its design for secondary memory.
- Structure:
  - The k-d-tree is a binary tree structure. Each node holds a point that divides space into two parts along a selected dimension. Traversing the tree, the dimensions are alternated such that nodes with the same depth level in the tree always use the same dimension to split the space.
  - Newly inserted data points follow the unambiguous path to a leaf node. This leaf node is then split into two new sub-regions using the data point and the dimension at this depth level. As with the Quadtree, the k-d-tree usually leads to an unbalanced tree. Again, it is possible to bulk load data points in such a way that a perfectly balanced k-d-tree results.
  - Variants:
    - Different splitting strategies were proposed (e.g., split in the middle, split at an arbitrary point)
    - The k-d-B-tree was designed for secondary storage and improved insertion to obtain a balanced tree for the representation on secondary storage.



6.2.4 Gridfile

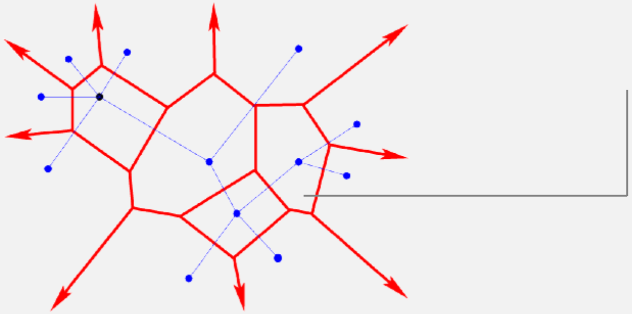
- Nievergelt and Hinterberger (ETH Zurich) developed in 1981/84 the Gridfile, a structure that was later extended with space-filling curves and found its way into many relational database extension to capture geo-information.
- Structure:
  - The data space is divided by a gridlines along each dimension. The resulting cells are numbered and indexed in a directory. Each directory entry points to a disk page that holds the data points of the corresponding cell. To save storage, several cells can share the same disk page. If a cell contains more points that fit into a disk page, a new gridline is added and a local re-organization of the cells and the directory become necessary.
  - Inserting and removal of data points is straightforward with the exception of over- and underflow of disk pages. Optimized bulk loading strategies can minimize the directory as well as the number of disk pages.
  - Note that the disk storage consumption does grow linearly with the number of data points while the directory grows super linearly (but not too fast). The grid acts like a quantization or hashing method for data points allowing the Gridfile to find data points with exactly one disk page load.





6.2.5 Voronoi Diagram

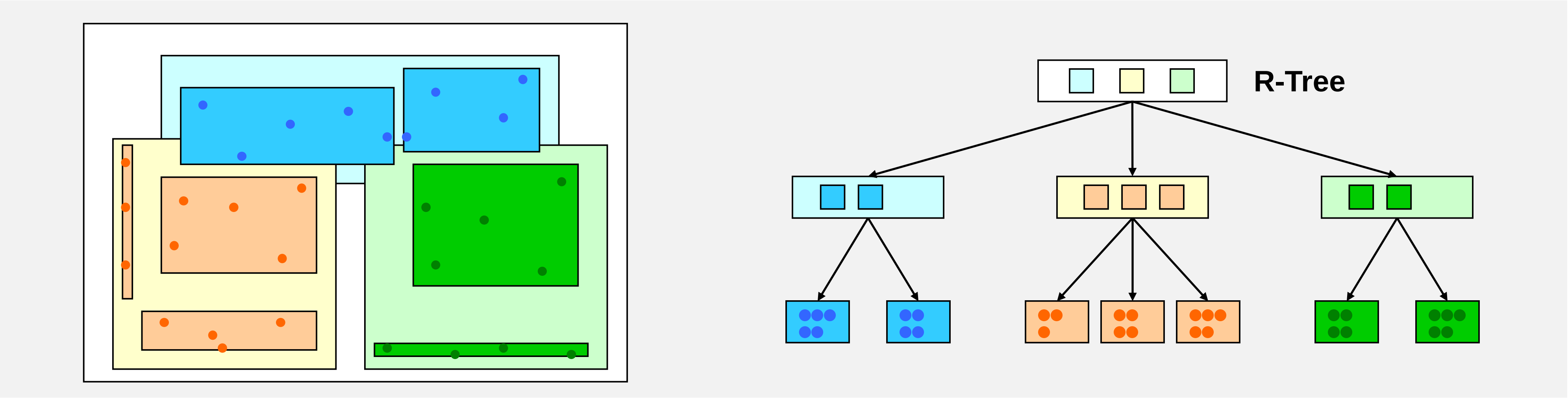
- Given a set of points, a Voronoi diagram partitions the space in such a way that each region holds exactly the sub-space that is closest to the same given data point. In other words, each region represents the pre-computed results of all nearest neighbor searches in the space.
- Structure:
  - Instead of storing the data points, we compute the Voronoi diagram and index the regions. To identify the nearest neighbor for a given query point, we identify the region it falls into and return the corresponding data points as its nearest neighbor. Voronoi diagrams can be computed for different distance metrics (not just Euclidean as in the picture below).
  - There are two fundamental challenges: computing the Voronoi diagram (especially in higher dimensional spaces) and storing the cells in such a way that we can quickly identify the containing cell of a data point. Consider the example below: some regions have six and more lines that define their shape. Storage consumption is known to grow exponentially with the dimensionality of the data points.
  - To reduce storage costs, we can approximate a Voronoi cell with its minimal bounding rectangle. Instead of a single region, we may have to look at several regions to identify the closest point.



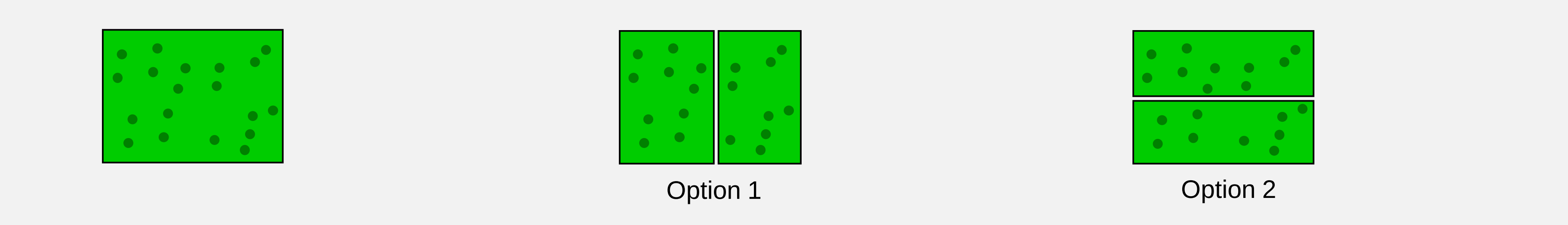
Voronoi Cell: all points in the cell are closest to blue center point

6.2.6 R-tree and variants

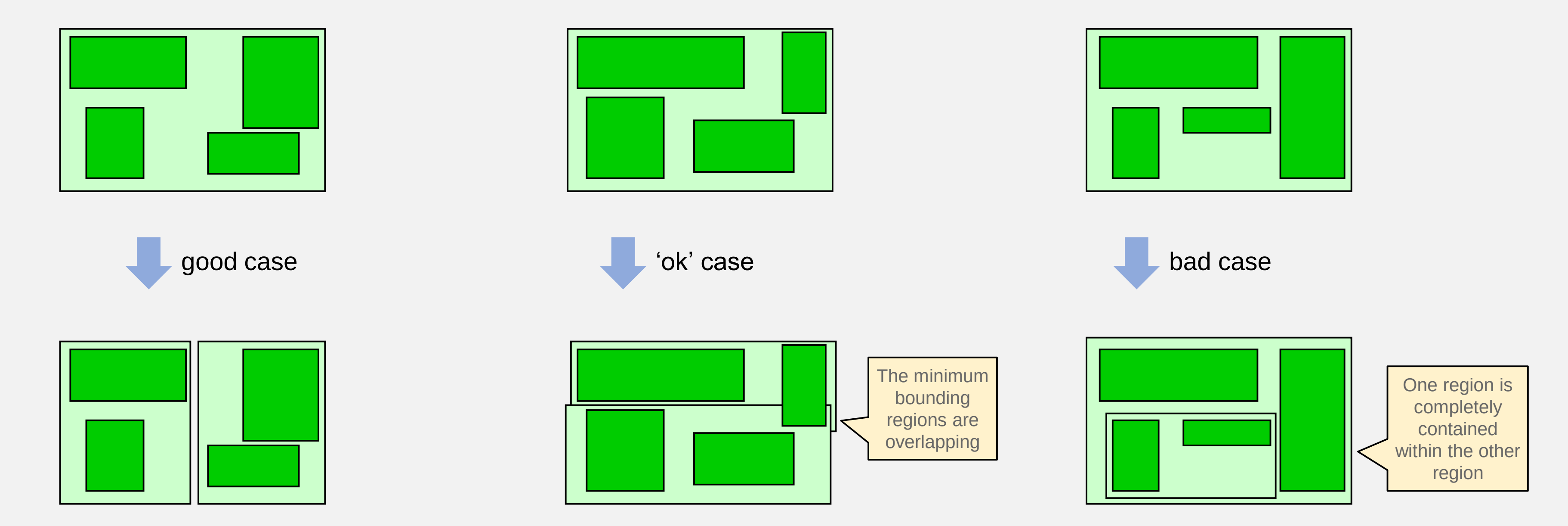
- Guttman described the R-tree in 1984 as an index structure for 2-dimensional data points. It was designed as a balanced tree with all leaves at the same level. In later years, numerous extensions have further optimized the structure and adapted it for higher dimensions.
- Structure:
  - The R-tree consist of two node types:
    - The **leaf nodes** hold the data points at the bottom of the tree. They are described by minimal bounding regions (MBR). The original R-tree used rectangles but other forms are possible.
    - The **inner nodes** hold a set of inner nodes or leaf nodes at higher levels of the tree. Again, a minimal bounding region encompasses the ones of all child nodes.
  - Insertion of a point follows the typical algorithm for balanced trees: a path is identified from the root to a leaf node (point must be contained in MBRs of nodes along the path). The point is inserted into the leaf node and, if the node, overflows, it is split and the two new leaf nodes are added to the parent node. The split may propagate all the way up leading to a new root.



- Splitting nodes:
  - Leaf nodes are straight forward: the data points are divided along a dimensions into two parts (select the median value of points along this dimension). The split guarantees that the two new leaf nodes do not overlap.



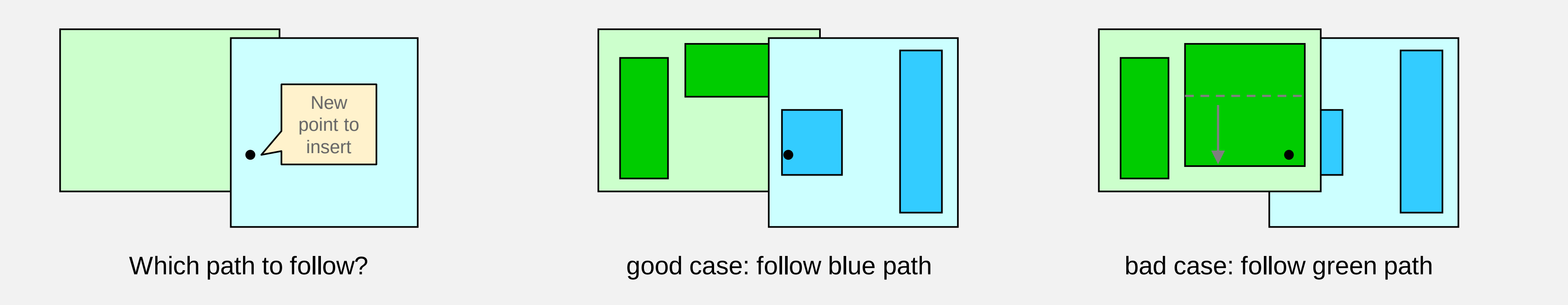
- Splitting an inner node is more difficult: the children are now minimum bounding regions which we may not separate perfectly into two halves. In bad cases, the minimum bounding regions of the two new inner nodes overlap.



The minimum bounding regions are overlapping

One region is completely contained within the other region

- Overlap is a problem: when searching for data points and if there is an overlap of regions, we have to follow several paths to find the point or ensure it does not exist. When inserting new points and if the data point falls into the overlap of regions, we must select a ‘best’ path. In bad cases, the overlap continues to grow (across leaf nodes) even if no new splitting occurs.



Which path to follow?

good case: follow blue path

bad case: follow green path

- R-tree Extensions: over the years, numerous extensions and optimizations for R-trees were published. Key aspects of optimizations include:
  - Shapes of minimum bounding regions, i.e., rectangles, spheres, combinations of shapes
  - Splitting: reduce overlaps of leaf nodes, re-insert points, rebuild tree
  - Size of nodes: increase page size if split is not beneficial
  - Metric Tree: no need for dimensional data but only requires a metric between objects
  - Examples: R+-Tree (1987), R*-Tree (1990), P-Tree (1990), TV-Tree (1994), vp-Tree (1994), GiST (1995), X-Tree (1996), SS-Tree (1996), SS+-Tree (1997), SR-Tree (1997), M-Tree (1997), Pyramid-Tree (1998), DABS-Tree (2000), P-Sphere Tree (2000), …and many more
- So what is so difficult about high-dimensional spaces?

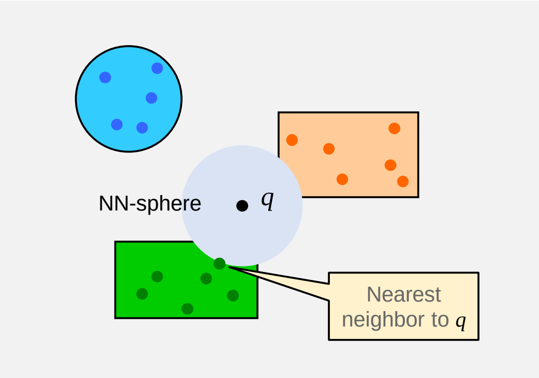
6.2.7 Nearest Neighbor Search Algorithm

- Hjaltson and Samet described a generic, optimal search algorithm to find the nearest neighbor in hierarchical structures. Optimal means that the number of visited nodes is minimal to prove correctness of the found nearest neighbor (cannot be done with less visits)
- The algorithm uses a priority queues for nodes and points. The priority corresponds to the distance of the query point to the data point or to the minimal bounding region. The queue is ordered by increasing distances. The algorithm works as follows for a given query object q

1. Initialization: the root node is added to the queue with the distance of its MBR to q
2. As long as the queue is not empty, fetch the top element of the queue → p
    a) If p is a data object, then p is the nearest neighbor to q
    b) If p is a leaf node, insert all contained data points with their distances to q
    c) If p is an inner node, insert all its child nodes with their distances to q

- Note that the algorithm only requires a distance measure between objects and between an object and a node (e.g., the minimal distance of a point to a rectangle, if the node is represented by an MBR of the form of a rectangle).
- Proof of correctness: The priority queue is organized by increasing distances. Due to the construction of nodes with minimum bounding regions, the children (nodes, objects) of a node must have equal or larger distances to the query object than their parent node. If a data object is at the top of the queue, then all nodes not yet visited must have a larger distance to the query object and recursively all their (grand-) children nodes and objects must lie farther away then the object at the top of the queue.

- Proof of optimality: assume we know the nearest neighbor to the query object q:
  - The circle around q through this nearest neighbor is the so-called Nearest Neighbor Sphere (NN-Sphere).
  - To proof correctness, all nodes that intersect with the sphere (i.e., lie closer to q than the nearest neighbor) must be considered by an algorithm as they may include a better answer. In the example on the right, the red rectangle must be considered but the blue circle is not needed to proof correctness. This set of nodes is also the minimal set of nodes that need to be checked.



  - The algorithm visits nodes in increasing order of their distance to q. If finally an object is at the top of the queue, this is the nearest neighbor. Only nodes with a smaller distance to q than the nearest neighbor were visited. Nodes with larger distances may be in the queue but not visited (they were added by their parent node).
- The algorithm works for any hierarchical structure and visits only (leaf) nodes that are required to prove correctness of the result. It is a generic implementation that only requires a distance measures but does not make any assumption about the internal structure except for hierarchy and that nodes encompass all their (grand-) children (which means that minimal distance to a node is smaller or equal to the minimal distance of any its child nodes and objects)
- Overlap of leaf nodes (as discussed in the R-tree) lead to a larger number of visited nodes in general. This is because query objects that lie in the overlap area of leaf nodes must always visit all these leaf nodes.

6.3 Curse of High Dimensionality

- While the methods for indexing multi-dimensional spaces work reasonably well for 2 to 5 dimensional spaces, it was observed that they quickly degrade as dimensionality increases. One of the common mistake made was the assumption that high-dimensional spaces just behave the same as low-dimensional space. However, our minds have difficulties to ‘imagine’ how a high-dimensional space behaves.
- In the following, we discuss the curse of high dimensionality. We want to understand why it is much more difficult to index high-dimensional spaces. We first consider a few peculiarities of high-dimensional spaces and then develop of mathematical understanding of what happens if we search for nearest neighbors in higher dimensions.

- **Assumptions:** To simplify the mathematical considerations, we assume a closed data space of the shape of a hyper cube $\Omega = [0,1]^d$. We further assume independent dimensions and uniform distributions along the dimensions (otherwise eliminate them with dimensionality reduction)

- **Observation:** Given $\Omega = [0,1]^d$, the probability that a data point lies inside a subspace is given by the volume of that subspace. The total volume of the space is 1 regardless of dimensionality.

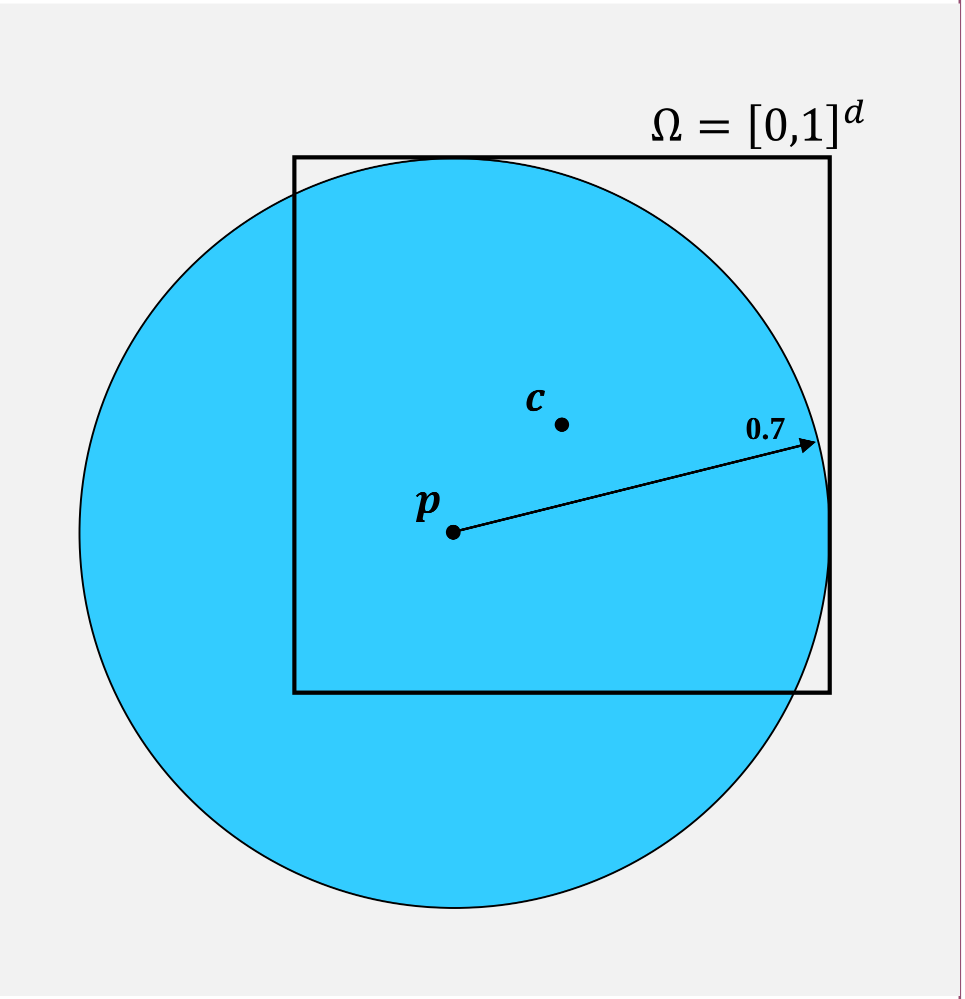
6.3.1 Peculiarity 1: Bad intuition for high-dimensional spaces

- Given
  - data space: $\Omega = [0,1]^d$
  - center of Ω: $\boldsymbol{c} = [0.5, \ldots, 0.5]$
  - a point $\boldsymbol{p} = [0.5, \ldots, 0.5]$
  - circle around $\boldsymbol{p}$ with radius 0.7

- In a 2-dimensional space (see figure), the circle covers most of the data space. It also follows that the center $\boldsymbol{c}$ of the data space lies inside this circle. In other words, circles with a radius of 0.7 are large and cover most of the data space Ω if their center lies within Ω.

- In higher dimensions, this is not the case:
  - Distance between $\boldsymbol{p}$ and $\boldsymbol{c}$ is $\delta = 0.2\sqrt{d}$
  - With $d > 12$, the distance is $\delta > 0.7$ and the center $\boldsymbol{c}$ is no longer inside the circle

  In very high-dimensional spaces, the circle still touches all sides opposite of $\boldsymbol{p}$ but it barely covers any space (volume goes to 0 with growing d)



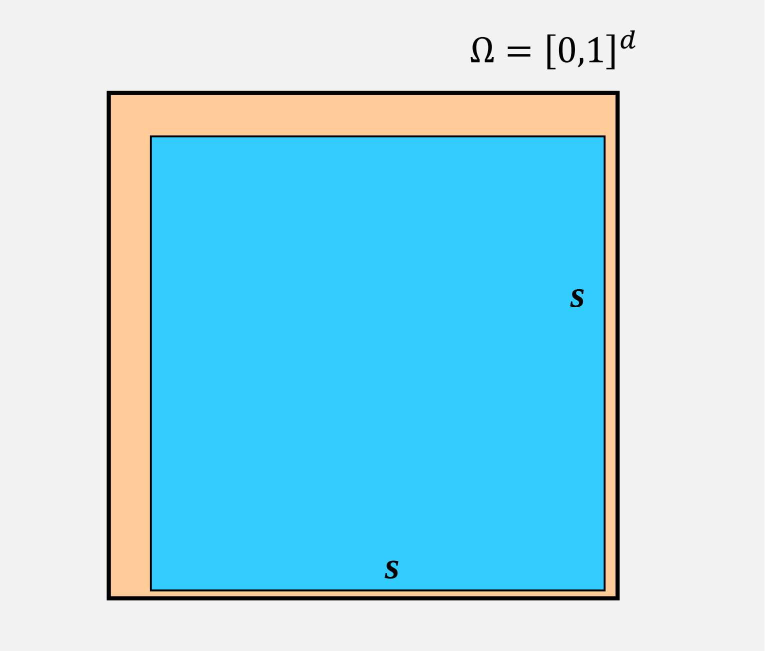
6.3.2 Peculiarity 2: Partitioning does not make sense

- Given
  - data space: $\Omega = [0,1]^d$
  - a limited number N of data points, e.g., $N = 10^9$
- In a 2-dimensional space, we can partition the data space by continuously splitting it along each axis into two halves (see gridfile, k-d-tree). The number of partitions doubles with each split, and we obtain $2^2 = 4$ partitions if all axes are split once.
- In higher dimensions, we still can split the data space into halves along an axis, but we can no longer split along each axis as the number of partitions grows exponentially with dimensionality. Consider the following table. Let d be the number of dimensions, N be the number of data points (e.g., $N = 10^9$). Then the number of cells M grows exponentially; if we split each dimension exactly once, then $M = 2\hat{}d$. Finally, $m = N/M$ is the expected number of points per cell.

dimensionality [d]	**# cells [$M = 2\hat{}d$]**	**# points [N]**	**# points per cell [$m = N/M$]**
10	$2^{10} = 1024$	10^9	$976'563$
50	$2^{50} = 1.12 \cdot 10^{15}$	10^9	$8.9 \cdot 10^{-7}$
100	$2^{100} = 1.27 \cdot 10^{30}$	10^9	$7.9 \cdot 10^{-22}$

- In higher dimensions, most of the cells are empty as the number of cells by far exceed the number of data points. The volumes of the cells becomes so small that it is even not likely that two points share the same cell (unless they are features from two extremely similar objects).

6.3.3 Peculiarity 3: Where are all the data points?

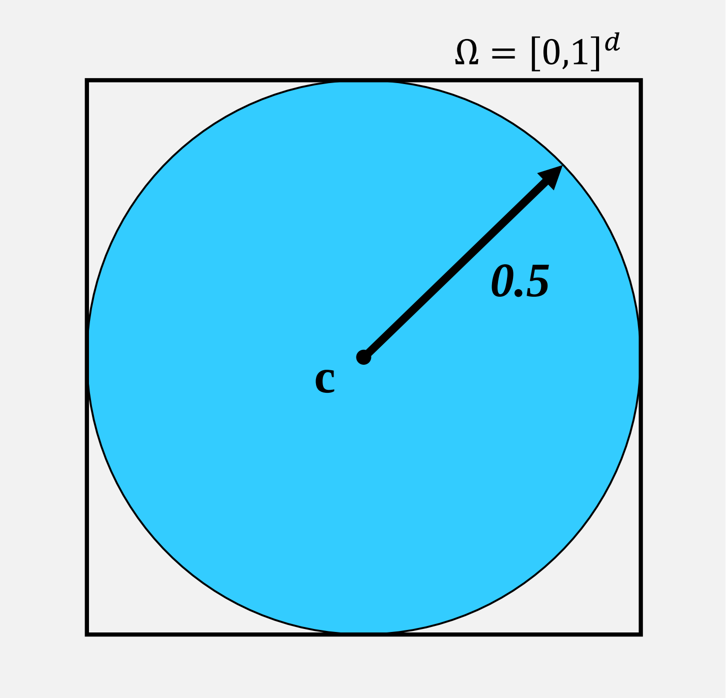


- Given
  - data space: $\Omega = [0,1]^d$
  - A hyper cube with side length $s = 0.95$
- Consider the right hand figure: Where are the data points more likely to be found? In the blue hyper cube with side length $s = 0.95$ or in the red area of Ω that is not covered by the hyper cube?

- In a 2-dimensional space, it is obvious that most points must fall into the blue area. Its volume is $s^2 = 0.90$ and thus 90% of the points are contained by the blue area.
- In a high dimensional space, most of the points lie in the read area! The volume of the blue hyper cube is s^d and with $s = 0.95$ this volume is shrinking exponentially to 0 as dimensionality grows:
  - with $d = 10$, still 60% of the points lie in the blue hyper cube and 40% are in the red area
  - with $d = 50$, only 8% of the points lie in the blue hyper cube and 92% are in the red area
  - with $d = 100$, only 1% of the points lie in the blue hyper cube and 99% are in the red area

- In other words, in higher dimensions, data points lie close to the edge of the data space. In at least one dimension, the value is either very close to 0 or very close to 1.

6.3.4 Peculiarity 4: The nearest neighbor is far away



- Given
  - data space: $\Omega = [0,1]^d$
  - a limited number N of data points, e.g., $N = 10^9$
  - center of Ω: $\boldsymbol{c} = [0.5, \dots, 0.5]$
  - circle around $\boldsymbol{c}$ with radius 0.5
- In a 2-dimensional space, we expect that the nearest neighbor is close to the query point (i.e., distances are small). The circle around $\boldsymbol{c}$ would surely contain the NN of $\boldsymbol{c}$.
- This is no longer true for high dimensional space! The volume of the d-dimensional sphere shrinks towards 0 with increasing d.
  - With $d = 10$**,** the volume is 0.002, i.e., 0.2% of the are inside the sphere
  - With $d = 100$**,** the volume is only $1.9*10^{-70}$, i.e., next to $\boldsymbol{c}$ with minimal probability no other point lies within the sphere.

  In higher dimensions, the circle around $\boldsymbol{c}$ must be much larger to contain the nearest neighbor. In other words, the distance between $\boldsymbol{c}$ and its NN have to be larger than 0.5.

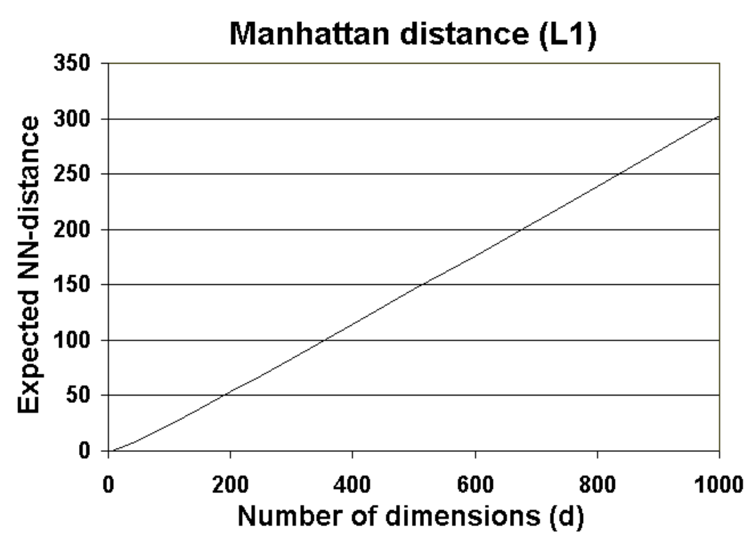
- So, how far away is the nearest neighbor in higher dimensional spaces?

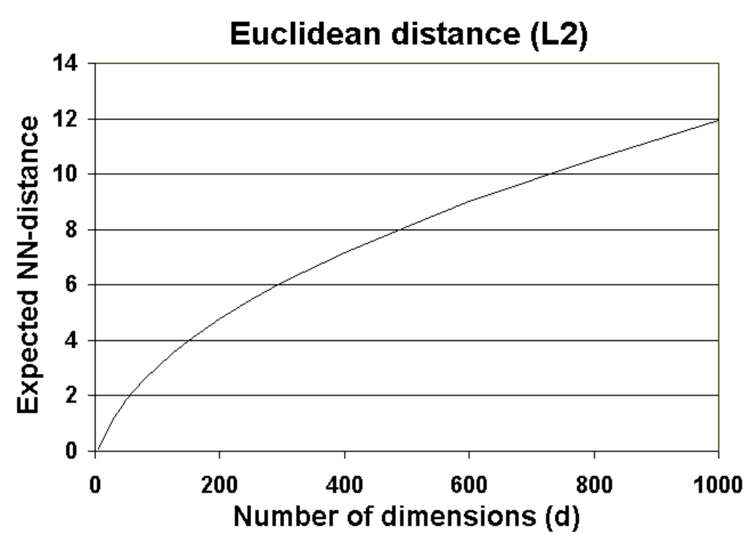
6.3.5 Cost Model for NN-Search

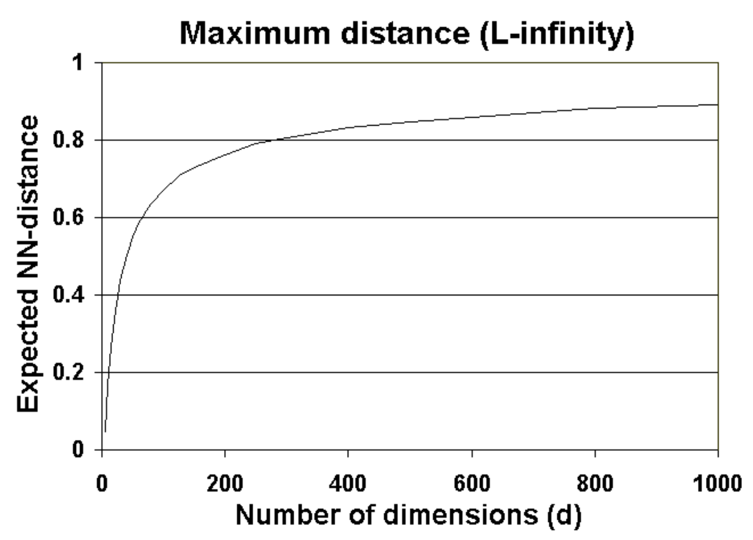
- In the following, we estimate the costs for NN-searches in hierarchical structures. To this end, we first have to determine the expected distance between query point and its nearest neighbor. Then we estimate how many leaf nodes, on average, are retrieved during the search.
  - Since we are using the optimal NN- search algorithm, we can easily determine the leaf nodes to be read: all the nodes that intersect with the NN-sphere around the query point.

- **Expected NN-distance**
  - The expected NN-distance is given as the average distance between a query point and its nearest neighbor
  - Basic idea:
    - For a given point in the data space and a radius r, determine the probability that the NN lies within the sphere around the point with radius r
    - With these probabilities, compute the expected value for r to obtain the expected NN-distance for the given point
    - Compute the mean expected NN-distance over all data points in the data space

- The expected NN-distance depends on the metric and the dimensionality of the space. The graphs below show the NN-distance for the metrics L_1, L_2 und L_∞

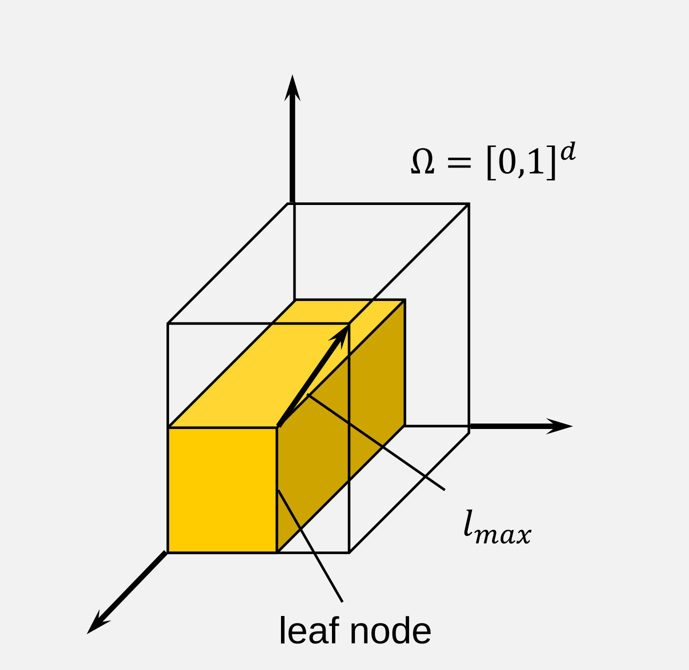




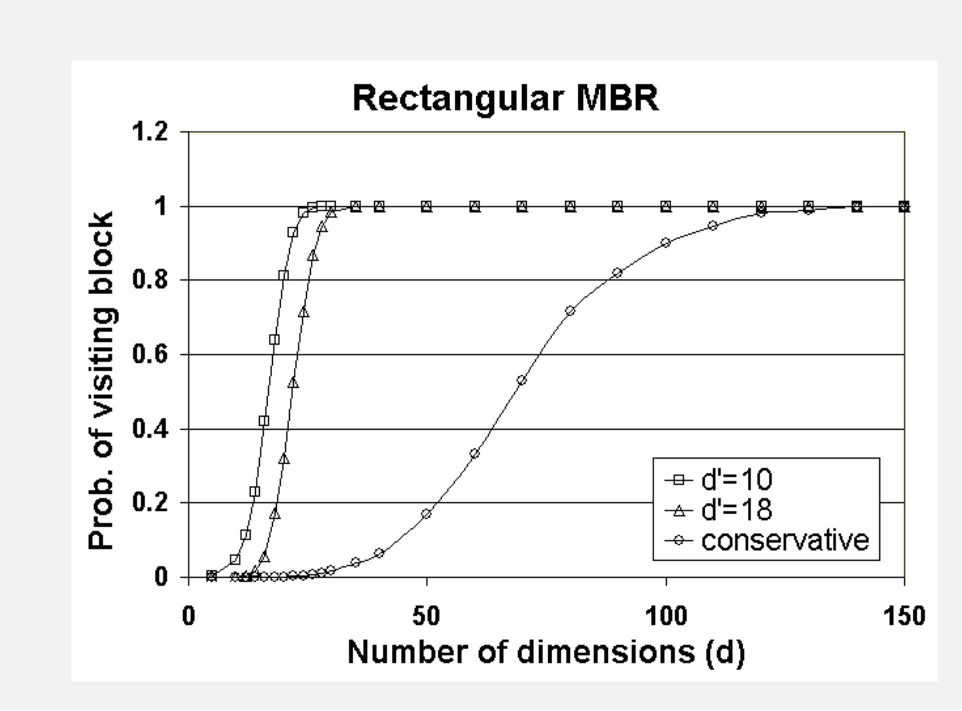


- **Number of leaf nodes to visit (simple consideration):**
  - Assume the tree uses rectangular MBRs. During splits, $d' < d$ axes were split. Further we assume that we always split in the middle. Hence, the MBRs of leaf nodes have the shape depicted in the figure below.
  - Let l_{max} be the maximum distance between a point in the space and a leaf node. Given the shape of the MBR, the distance is given by: $l_{max} = 0.5 \cdot \sqrt{d'}$
  - If we compare this distance with the expected distance, we obtain the following surprising results (see table):
    - with $d = 40$, l_{max} is about the expected NN-distance, with $d = 100$, l_{max} is much smaller than the expected NN-distance. This is because of the limited number of splits we can perform (ensuring we have non-empty leaves)
    - with $l_{max} < \mathrm{NN} - \mathrm{dist}$, each query points lies closer to all leaf nodes than to its NN. Hence, the MBR of each leaf intersects with the NN-sphere
    - thus, an optimal NN-search must visit all leaves to find the nearest neighbor to any point in the data space
  - Why do we use a hierarchical structure if we need to read all data anyway?

d	***N***	***d'***	l_{max}	$\mathrm{NN} - \mathrm{dist}$
40	10^6	14	1.87	1.80
100	10^6	15	1.94	3.00

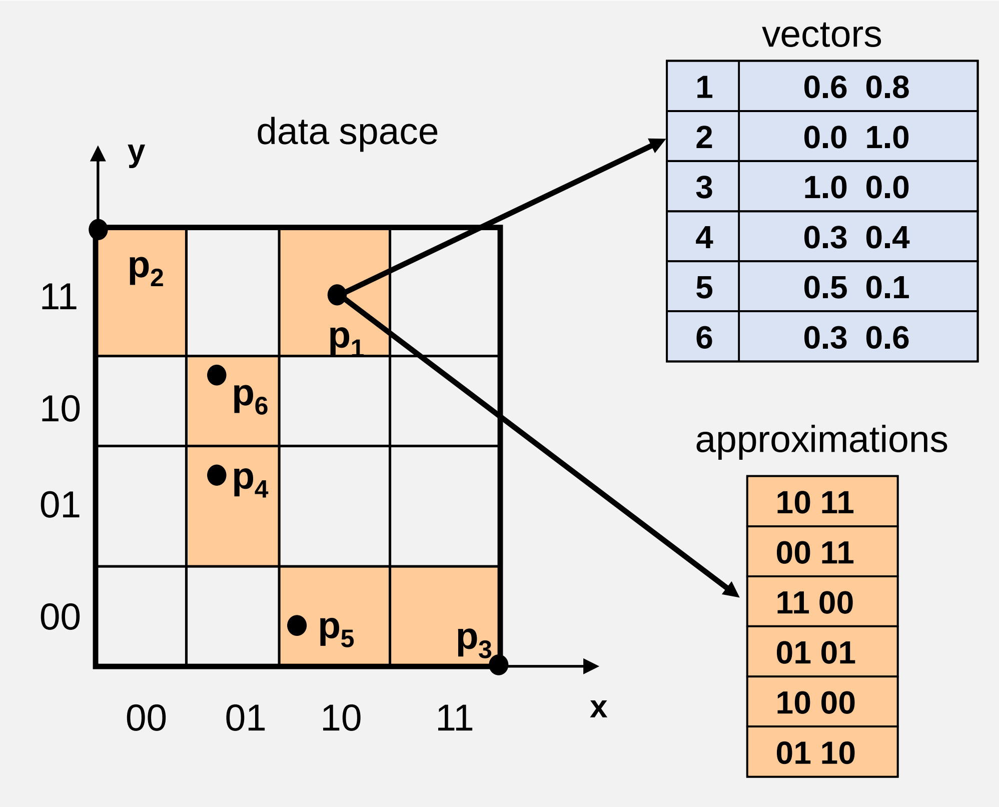


- **Simple cost model – when is it better to use brute-force NN-search**
  - With spinning disks, access to leaf nodes results in a random access pattern on the disk which is much slower than a sequential read pattern. To be efficient, we do not want to read more than 10% of the leaf nodes; otherwise, a sequential scan through all data is faster. With SSD, the threshold can be higher (no penalty on random access). Still, to be faster (and justify the use of a complex structure), we want to read significantly smaller amounts of leaf nodes.
  - The figure on the right shows the percentage of leaf nodes that are required to determine the nearest neighbor. The graphs “d’=10” and “d’=18” show the percentage of leaf nodes for rectangular MBRs with splitting 10 and 18 dimensions, respectively. The graph “conservative” is an optimized structure with leaf nodes that contain only two points (which are nearest neighbor to each other). Such a structure has the minimal size for MBR, yet it still degenerates with dimensions above 100
  - To be fast, the percentage of leaf nodes to be read should be well below 10%. This limits hierarchical methods to less than about 10 dimensions.
  - Note that real data often has correlations between dimensions. In such cases, the limits apply to the “true” dimensionality of the underlying data. As a result, it is possible that hierarchical methods work in higher dimensions. But in such cases, it would be better to apply a dimensionality reduction to eliminate correlations.



6.3.6 The Vector Approximation File (VA-File)

- The Vector Approximation File (VA-File) was developed in 1997 at ETH Zurich. Its aim was to accelerate the sequential scan through the data set through quantization. Typical speed-ups of factor 4-8 over a brute-force method made it the fastest NN-search algorithm.
- The VA-File reverses the curse of high dimensionality to benefit from it. At its core and as shown with the figure below, it quantizes the vectors with lower precisions (e.g., using 4-8 bits instead of 32/64 bits per dimension). The quantization error is rather small:
  - Consider a high dimensional space (d>50) and a data space with values in $\Omega = [0,1]^d$
  - If we use 8 bits per dimension, values are quantized to the closest $1/256$ steps leading to an average error of $1/512$
  - Quantization error is rather small: the volume of the area with the same quantized representation shrinks with $(1/256)^d$ and thus quickly to zero as dimensionality d grows. With other words: the area becomes so small that quantization is unlikely to map two points to the same representation.
  - Distance errors are proportional to the quantization error and grow at the same rate with dimensionality as the expected NN-distance.



vectors

1	0.6 0.8
2	0.0 1.0
3	1.0 0.0
4	0.3 0.4
5	0.5 0.1
6	0.3 0.6

approximations

10 11
00 11
11 00
01 01
10 00
01 10

- The VA-File can operate in two modes
  - Use only approximations to produce "a good enough" answer for the nearest neighbor search. Error rates are small and if used on real features it cannot be noticed due to the fuzzy definition of similarity. Note that features themselves are an approximation of the signal information and are sensitive to small changes (e.g., illumination, sampling rate). So we have no “absolute” true position of an object in the features space but more a region to which the object is mapped
  - Produce correct results with a 2-phase filtering approach.

Phase 1:

- $lbnd_i \leq \delta_i \leq ubnd_i$ defines the bounds for a distance between a query point (query is not quantized) and the rectangular region spanned by the quantized approximation of p_i
- If $lbnd_j > ubnd_i$, it follows that $\delta_j \geq lbnd_j > ubnd_i \geq \delta_i$ and thus $\delta_j > \delta_i$ and data point p_j cannot be the nearest neighbor as data point p_i must lie closer to the query point
- The above filtering is very effective due to the small ranges $[lbnd_i, ubnd_i]$. Typical results are that 99% of points can be excluded just with this filtering step

Phase 2:

- To identify the correct NN, we compute true distances between the query point and the candidates left from Phase 1. We do this in increasing order of their lower bounds, i.e., the point with the smallest lower bound is the most likely candidate to be the nearest neighbor
- We can stop as soon as we have found a true distance δ_i such that for all remaining lower bounds $lbnd_j$ it holds: $\delta_i < lbnd_j \leq \delta_j$
- This filtering step is effective and only 0.001% of the points are considered for distance computation

- The key benefit of the VA-File is to reduce the amount of data to be read. Given the quantization scheme, it is possible to accelerate distance computations with precomputed (squared) differences along all dimensions for a current query point. This provides a significant acceleration for in-memory search (no multiplications for Euclidean distances required, only additions)
- Given the sequential structure, it is also much simpler to evaluate complex queries over several features if the feature files are sorted in the same way.

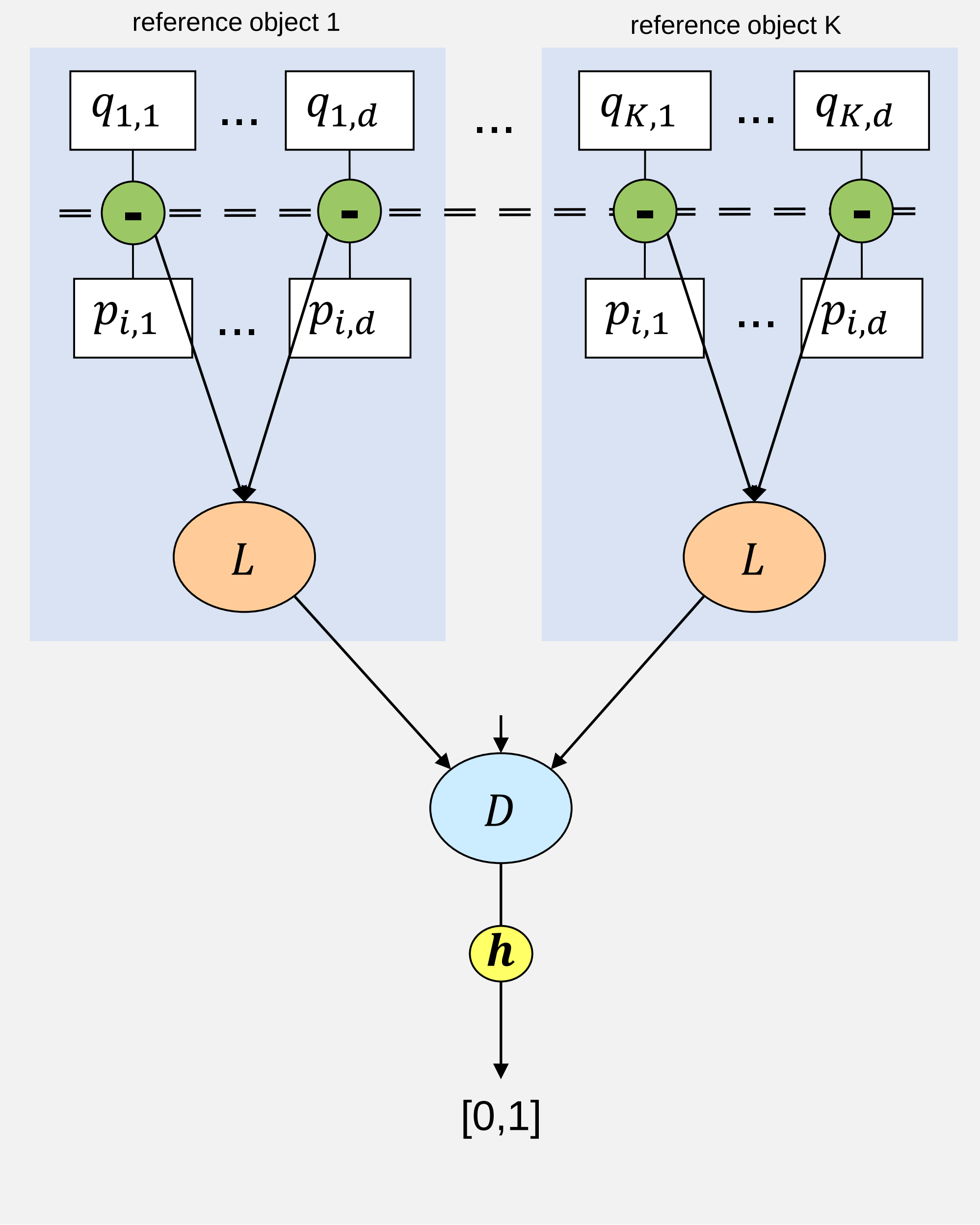
6.4 Complex Similarity Search

- In the following, we consider three extensions to the similarity search
  - queries with multiple reference objects
  - queries using different features (e.g., text and color)
  - queries using features and predicates (e.g., large image)
- Example queries:
  - All audio files similar to song B and song C
  - All images that are similar to image A according to color and shape
  - All images similar to image D that contain the terms „dolphin“ and „wale“
  - All video clips similar to clip E which are for free (price=0) and contain a „car“

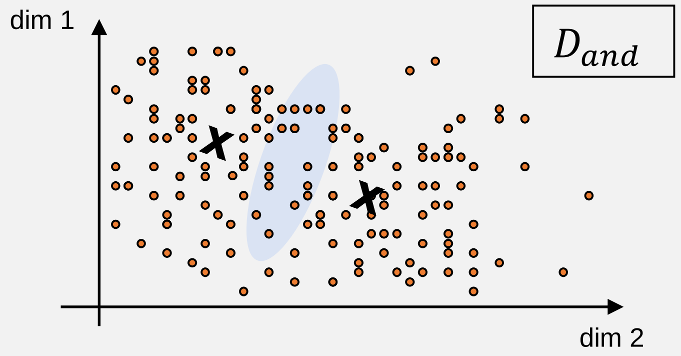
- We first discuss the evaluation schemes to compute a score given a complex queries. Then, we look into methods that help to evaluate such queries. While multi-reference queries are straightforward, multi-features and predicate search require additional methods.

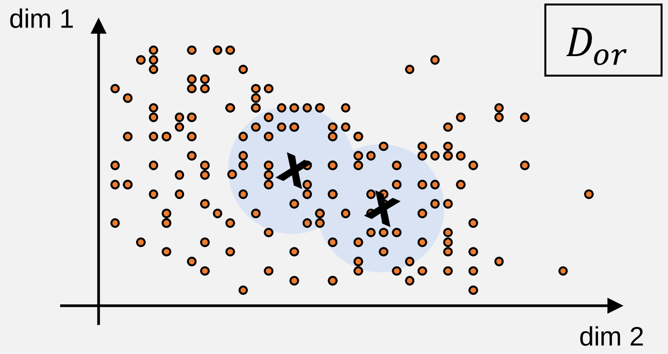
6.4.1 Multi-reference Queries

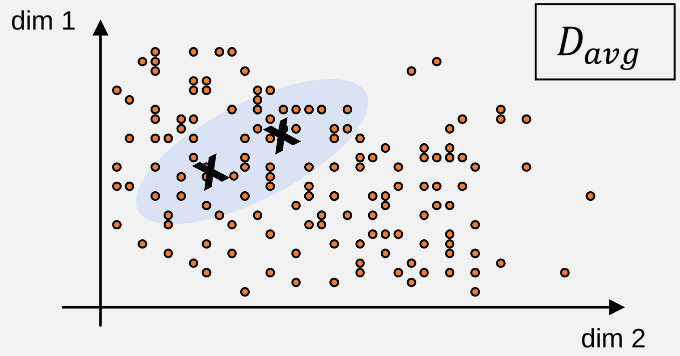
- A multi-reference query takes two or more reference objects for the search. Let $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_K$ be the features of the K reference objects
- Note: we consider only a single feature
- The right hand diagram shows the evaluation scheme from the top (feature vectors) to the bottom (score) for the i-th object with feature representation $\boldsymbol{p}_i$
  - The top part (blue boxes) evaluates a distance between the features $\boldsymbol{p}_i$ of the i-th object and $\boldsymbol{q}_k$ of each reference object. We use the distance measure L as defined for the feature.
  - The bottom part combines distances with a distance combination function D and applies a correspondence function h to map distance to scores. Examples for distance combining functions are:
    - D_{and}: $\delta(\boldsymbol{p}_i) = \max_k L(\boldsymbol{p}_i, \boldsymbol{q}_k)$
    - D_{or} : $\delta(\boldsymbol{p}_i) = \min_k L(\boldsymbol{p}_i, \boldsymbol{q}_k)$
    - D_{avg}: $\delta(\boldsymbol{p}_i) = \frac{1}{K}\sum_k L(\boldsymbol{p}_i, \boldsymbol{q}_k)$



- **Interpretation:** assume we have two reference objects in a 2-dimensional feature space:
  - Each reference object spans a neighborhood in the feature space that contain similar objects
  - Depending on the query semantics, the neighborhoods are merged in different ways:
    - AND-semantics: all reference objects must be matched as good as possible, i.e., the maximum distance to a reference object must be as small as possible. This corresponds to the intersection of the neighborhoods around the reference objects
    - OR-semantics: object must be close to at least one reference object, i.e., the minimum distance to a reference object must be as small as possible. This corresponds to the union of the neighborhoods around the reference objects
    - AVERAGE-semantics: all reference objects matters and we need a good compromise, i.e., the sum of distances to reference objects must be as small as possible. This corresponds to an elliptical area with the two reference objects being its focal points
  - Other semantics and definitions for a distance combining function are possible







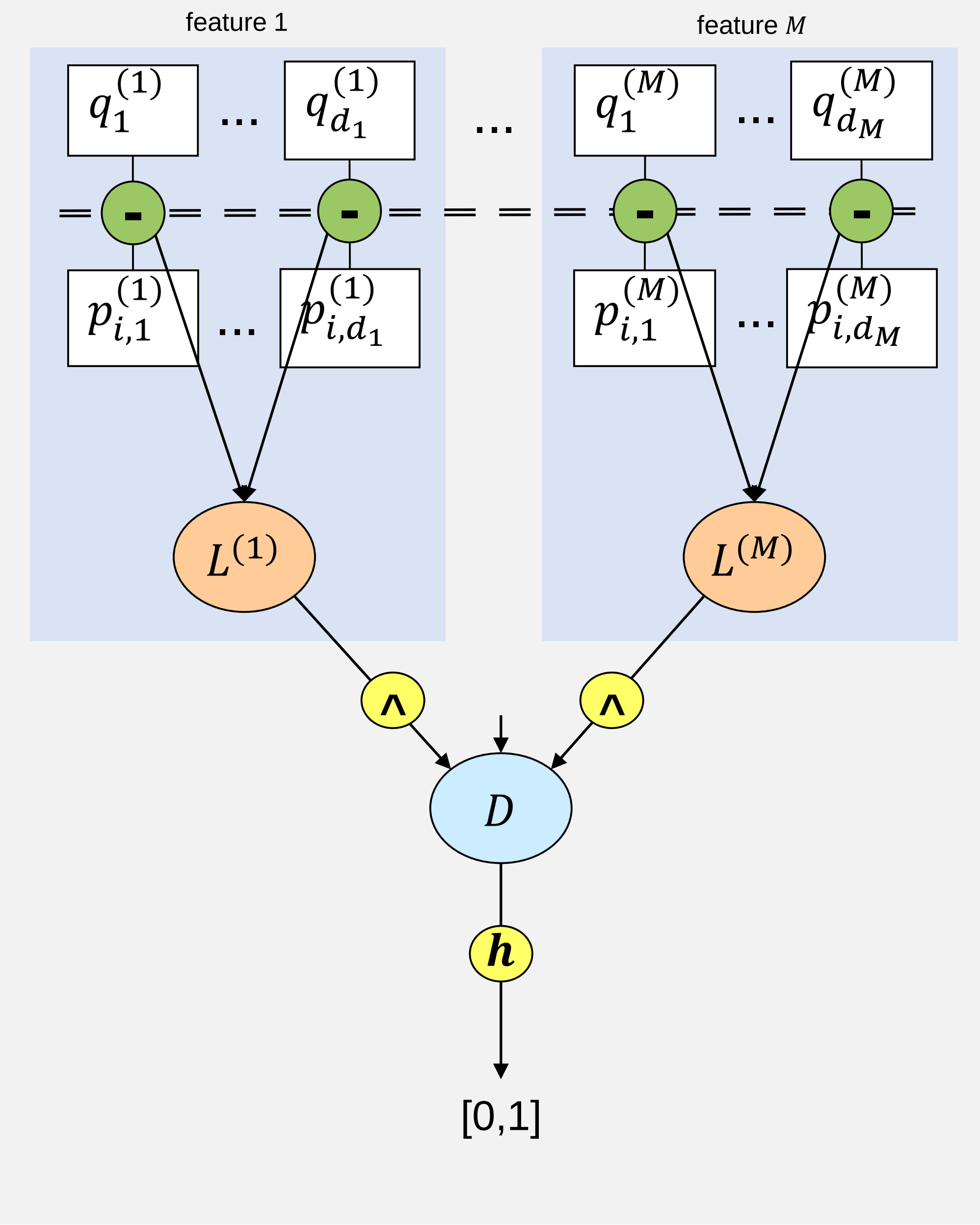
- **Evaluation:** Ciaccia (1998) demonstrated how to evaluate queries with several reference objects
  - In order to be applicable, the distance combining function must be monotonic

    $$x > x' \text{ and } y > y' \colon D(x,y) > D(x',y')$$

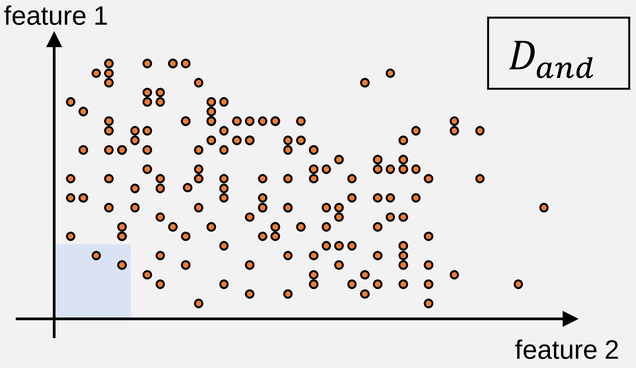
    This is not a hard constraint but rather natural: if the distance x and y are larger than their counterparts x' and y', then the combined distance for x and y also must be larger than the one of x' and y'. All the functions we considered before fulfill this constraint.
  - The evaluation of multi-reference objects is an extension of the underlying search algorithm (e.g., the optimal NN-search algorithm or the brute-force scanning with the VA-File): whenever a distance between query and data point, a lower/upper bound on such a distance, or distances to minimum bounding regions is required, then replace that function with
    - compute distances/bounds to each reference object
    - combining the distances/bounds with the distance combining function
    - continue this value in the algorithm
  - In other words, only the function to compute distances changes but the algorithm stays the same. It is straightforward to write such a generic NN-search algorithm for all the methods we considered so far.
  - The proof of correctness is straightforward based on the constraint above. Monotonicity ensures that, for instance, bounds are still under/over estimating real distances and that the stop criteria in the NN-search of Hjaltson an Samet still holds true.
  - Experience shows that the number of reference objects does have a small impact (next to additional computational efforts for distances) on search performance: the more reference objects we select, the more data points/leaves need to be visited to find the nearest neighbor

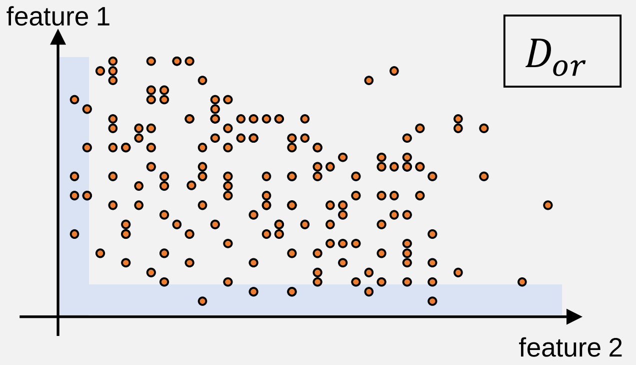
6.4.2 Multi-feature Queries

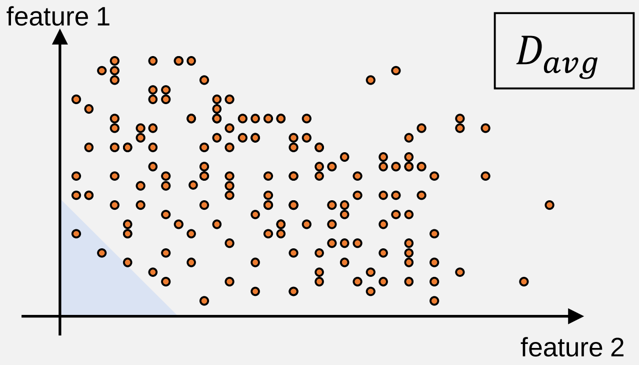
- A multi-feature query asses similarity values with two or more features. Let $\boldsymbol{q}^{(1)}, \dots, \boldsymbol{q}^{(M)}$ be the M features of the reference object
- The right hand diagram shows the evaluation scheme from the top (feature vectors) to the bottom (score) for the i-th object with feature representations $\boldsymbol{p}_i^{(1)}, \dots, \boldsymbol{p}_i^{(M)}$
  - The top part (blue boxes) evaluates a distance between $\boldsymbol{p}_i^{(j)}$ and $\boldsymbol{q}^{(j)}$ for each of the feature. We use the distance measure $L^{(j)}$ as defined for the feature j.
  - The bottom part combines normalized distances (^-function) with a distance combination function D and applies a correspondence function h to map distance to scores. Examples for distance combining functions are:
    - D_{and}: $\delta(\boldsymbol{p}_i) = \max_j \widehat{L^{(j)}}(\boldsymbol{p}_i^{(j)}, \boldsymbol{q}^{(j)})$
    - D_{or} : $\delta(\boldsymbol{p}_i) = \min_j \widehat{L^{(j)}}(\boldsymbol{p}_i^{(j)}, \boldsymbol{q}^{(j)})$
    - D_{avg}: $\delta(\boldsymbol{p}_i) = \frac{1}{M}\sum_j \widehat{L^{(j)}}(\boldsymbol{p}_i^{(j)}, \boldsymbol{q}^{(j)})$



- **Interpretation:** assume we have two features
  - Each feature has its own space and may have different dimensionality. Unlike with multi-reference queries, we cannot visualize the combined space. Instead, the diagrams on the right show the data objects in a two dimensional diagram with each dimension corresponding to the distance between data objects and query object along a feature
  - Depending on query semantics, the shown areas contain the best matches:
    - AND-semantics: distances for both features must be small, i.e., the maximum distance over features must be as small as possible. This is the area close to the left lower corner
    - OR-semantics: distances for at least one feature must be small, i.e., the minimum distance over features must be as small as possible. This corresponds to union of the areas that lie close to an axis.
    - AVERAGE-semantics: distances of both features matter and contribute to the result, i.e., the sum of distances for the features must be as small as possible. This is the triangle shaped area in the left lower corner.
  - Other semantics and definitions for a distance combining function are possible







- **Evaluation:**
  - Before we can apply a distance combining function, we need to normalize the distances:
    - Note that this was not necessary with multi-reference queries as the sub-queries compute distance in the same space and hence already yield comparable distances
    - With multi-feature queries and if the features come from different spaces, distances can vary greatly from feature to feature. If we do not apply a normalization function, a distance combining function like D_{or} is likely to prefer one feature over others, if that one feature yields smaller distances.
    - Normalization includes Gaussian normalization (estimate mean and standard variance through sampling per feature) or a min-max normalization. Other normalization schemes that make distances comparable are feasible as well
  - For index structures with sequential organization, we can order data points in each feature file in the same order. As we scan through all feature files in parallel, we can apply the same method as discussed for multi-reference queries
  - If at least one feature is not organized sequentially, we need to somehow combine the results of sub-queries over a single feature. Consider the search for text with an inverted file and the search for color with an R-tree.
    - OR-semantics: if distances are made comparable, first compute all queries for each feature and then take the object with the smallest distance for any of the features
    - This approach is not possible for AND-semantics and AVERAGE-semantics. In such cases, we need to scan through the results of all sub-queries until we find a data objects for which we can prove that it is better than all other objects. Fagin has defined in 1996 a generic algorithm that can handle queries with arbitrary distance combining functions as long as monotonicity applies.

- Fagin (1996): given a monotonic distance combining function D fulfilling

$$x > x' \text{ and } y > y'\text{: } D(x,y) > D(x',y')$$

  - The algorithm operates in three phases:
    - **1st Phase (sorted access):** each sub-query produces a stream of data objects ordered by increasing distances. We read from each stream a number of elements and all the objects retrieved form the set $\mathbb{A}$. Let $\mathbb{L}$ be the set of objects that was returned for all sub-queries. As soon as $\mathbb{L}$ is non-empty, the first phase ends (for k-NN search we stop if $\mathbb{L}$ contains k objects).
    - **2nd Phase (random access):** Determine for each object in $\mathbb{A}$ the remaining unknown distances for all sub-queries
    - **3rd Phase (computation):** Compute overall distances with the distance combining function D for all objects in $\mathbb{A}$ and return the object with the smallest overall distance

color	shape	texture
a, 0.4	a, 0.6	d, 0.1
b, 0.5	d, 0.7	b, 0.2
c, 0.7	b, 0.8	e, 0.7
d, 0.9	z, 0.9	z, 0.8
...	...	...

1st Phase: read the first three entries of each stream; b was returned by all streams
$\mathbb{L} = \{b\},\ \ \mathbb{A} = \{a, b, c, d, e\}$

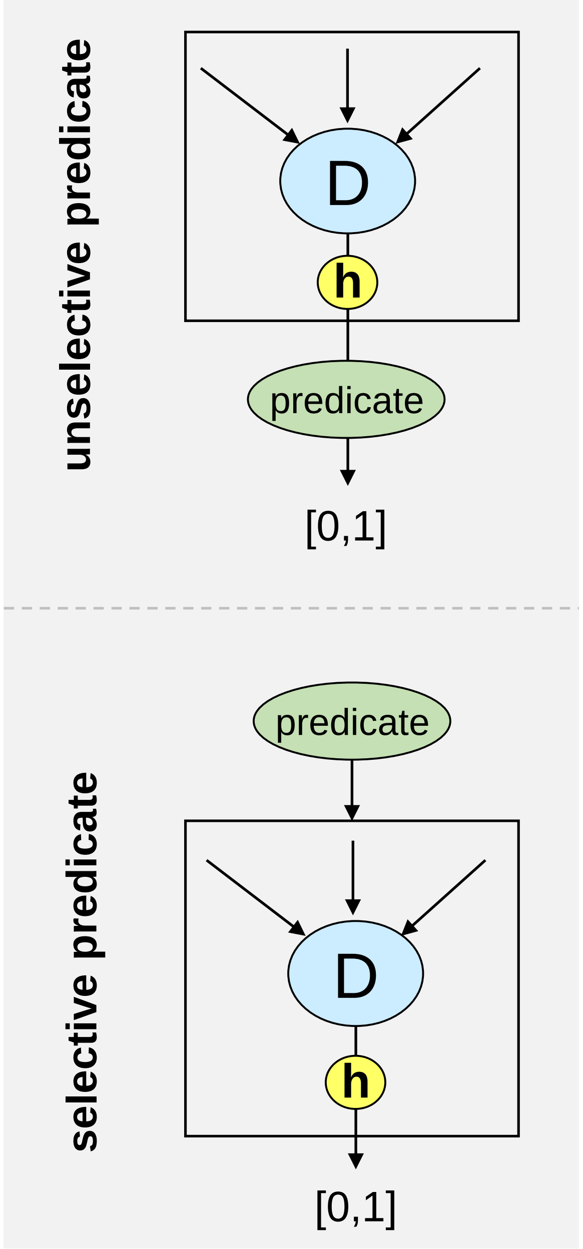
2nd Phase: determine missing distances of objects in $\mathbb{A}$.

3rd Phase: compute overall distances and determine the best object from $\mathbb{A}$.

- Proof of correctness: as soon as we have found an object o in the result stream for all sub-queries ($\mathbb{L} = \{o\}$), we have a first overall distance using the distance combining function.
  - For objects in $\mathbb{A}$, we do not know yet in Phase 1 whether they are better or worse than the object in $\mathbb{L}$; hence, we compute all remaining distances for objects in $\mathbb{A}$ and return the best
  - For all objects not in $\mathbb{A}$, we know that their distances must be larger for all features than for the object o. Due to the monotonicity constraint for distance combining function, we also know that their overall distance must be larger than the one of o. Hence, we can safely exclude them from further calculations.
- QuickCombine from Güntzer (2000) is an extension of Fagin's algorithm which reduces the number of computations in the 2nd and 3rd phase.
  - For objects in $\mathbb{A}$ (and not in $\mathbb{L}$), we need to evaluate all missing distances and apply the distance combining function. Assume $\mathrm{a} \in \mathbb{A}$ has no distance for feature j. Instead of computing the real distance, we can first use the largest distance for feature j as seen in its stream. This is a lower bound on the real distance for feature j (a follows later in the stream and must have a larger distance as the stream is ordered by increasing distances).
  - Applying the monotonicity constraint for distance combining function, we can compute a lower bound on the real overall distance. If this distance is larger then the currently best overall distance, we can dismiss the object a. If the distance is smaller, then a remains a candidate and we need to compute real distances for each feature
  - QuickCombine further optimizes the order to read from the streams to increase the pruning of candidates in phase 2 and 3. Evaluations have shown a speed-up of a factor of 100
- The algorithms work sufficiently well under "good" conditions. It is, however, possible to construct examples where large fractions of objects are in $\mathbb{A}$ with little options to prune. The algorithm performs sub-linearly in the number of objects, but easily can take several minutes to compute.

6.4.3 Queries with Predicates

- Queries with predicates filter the results and let only objects pass through that fulfill the predicate. In other words, if the predicate evaluates to false, the score is set 0, otherwise the score is passed on as is
- There are two evaluation schemes depending on how selective the predicate is
  - If the majority of the objects passes the predicate test, the best approach is to evaluate predicates as a last step (see right upper image). The scores are adjusted based on the outcome of the predicate test. Since the predicate is not selective, we may have to go through a few objects only to find the best match that fulfills the predicate
  - If a minority of the objects passes the predicate test, we better first evaluate the set of objects that fulfill the predicate (e.g., SQL query against the meta data of objects). Then we compute similarity scores for each of these objects. Since the predicate is not selective, applying the test at the end is not effective as we have to produce hundreds (or even thousands) of results before a first object fulfills the predicate
- We can also consider Boolean retrieval over terms as a special form of a predicate



6.4.4 Weighting Features and Reference Objects

- When composing a complex query, we want to weigh individual features and reference objects differently. However, weighting is only straightforward with average distance combining function but not with maximum/minimum:
  - Consider two features with normalized distances with a mean value of 10 and a standard deviation of 1. That is, almost all distances lie between 6 and 14.
  - Let us now weigh feature 1 with 0.8 and feature 2 with 0.2. Distances of feature 1 now range between 4.8 (6*0.8) and 11.2 (14*0.8), while distances of feature 2 range between 1.2 (6*0.2) and 2.8 (14*0.2). What happens if we apply the maximum and minimum function?
  - If we apply AND-semantics (maximum), feature 2 dominates the results. Since its distances are mostly above 4.8 and distances for feature 1 seldom exceed 2.8, the score is in almost all cases derived only from feature 2. It is as if feature 1 is not taken into account at all. However, our weighting does prefer feature 1 but not to such an extend
  - If we apply OR-semantics (minimum), feature 1 dominates the results. This is because distances for feature 1 are rarely below 4.8 while the distance for feature 1 are mostly below 2.8. In this case, feature 2 does not contribute much to the result, contrary to our intention to give it more weight than feature 1.
- From these observations, we conclude that, in general, the weighted distance combining function $D^{\boldsymbol{w}}(x_1, \ldots, x_K)$ with weights $\boldsymbol{w}$ cannot be simply derived from its unweighted form with weighted input distances:

$$D^{\boldsymbol{w}}(x_1, \ldots, x_K) \neq D(x_1 \cdot w_1, \ldots, x_K \cdot w_K)$$

- Fagin (1997) described a generic way how to solve the dilemma; his approach works for any distance combining function that fulfills the monotonicity constraint.
  - Fagin postulated that the weighted combining function $D^{\boldsymbol{w}}(x_1, \dots, x_K)$ must fulfill the monotonicity constraint and, in addition, must be steady if distances or weights change (no "jumps")
  - Without loss of generality, distances are ordered by increasing value of their weights. Furthermore, we add a sentinel weight $w_{K+1} = 0$ to simplify the formula.
  - Fagin was showing that the only weighted distance combining function that fulfills the above criteria is given as follows:

$$D^{\boldsymbol{w}}(x_1, \dots, x_K) = \sum_{i=1}^{K} i \cdot (w_i - w_{i+1}) \cdot D(x_1, \dots, x_i)$$

Written out

$$D^{\boldsymbol{w}}(x_1, \dots, x_K) = (w_1 - w_2) \cdot D(x_1) + 2 \cdot (w_2 - w_3) \cdot D(x_1, x_2) + \cdots + K \cdot w_K \cdot D(x_1, \dots, x_K)$$

  - Let us now apply it to our example from before with weights $\boldsymbol{w} = [0.8, 0.2]$
    - AND: $D^{\boldsymbol{w}}_{max}(x_1, x_2) = (0.8 - 0.2) \cdot \max(x_1) + 2 \cdot 0.2 \cdot \max(x_1, x_2) = 0.6 \cdot x_1 + 0.4 \cdot \max(x_1, x_2)$
    - OR: $D^{\boldsymbol{w}}_{min}(x_1, x_2) = (0.8 - 0.2) \cdot \min(x_1) + 2 \cdot 0.2 \cdot \min(x_1, x_2) = 0.6 \cdot x_1 + 0.4 \cdot \min(x_1, x_2)$
    - AVG: $D^{\boldsymbol{w}}_{avg}(x_1, x_2) = (0.8 - 0.2) \cdot x_1 + 2 \cdot 0.2 \cdot \frac{x_1 + x_2}{2} = 0.8 \cdot x_1 + 0.2 \cdot x_2$
  - As we see, in all forms x_1 and x_2 can contribute to the overall result. With OR/AND semantics, x_1 is always taken into account, while x_2 only contributes if it is smaller/larger than x_1.

6.5 References

- **Is NN-search meaningful in high-dimensional spaces**

  - S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. **A Cost Model For Nearest Neighbour Search**. In Proceedings of the ACM Symposium on Principles of Database Systems (PODS), pages 78-86, Tucson, Arizona, USA, May 1997. ACM Press.
  - R. Weber, H.-J. Schek, and S. Blott. **A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces**. In Proceedings of the International Conference on Very Large Databases (VLDB), New York, USA, August 1998.
  - K. Beyer, J.Goldstein, R. Ramakrishnan, and U. Shaft. **When is ``nearest neighbour'' meaningful?** In Proc. 7th Int. Conf. Data Theory, ICDT, number 1540 in Lecture Notes in Computer Science, LNCS, pages 217-235. Springer-Verlag, 10-12 January 1999.
  - A. Hinneburg, C. C. Aggarwal, and D. A. Keim. **What Is the Nearest Neighbor in High Dimensional Spaces?** In Proceedings of the International Conference on Very Large Databases (VLDB), pages 506-515, Cairo, Egypt, Sept. 2000. Morgan Kaufmann.
  - C. C. Aggarwal, A. Hinneburg, and D. A. Keim. **On the surprising behavior of distance metrics in high dimensional spaces**. In Proceedings of the International Conference on Database Theory (ICDT), volume 1973 of Lecture Notes in Computer Science, pages 420-434, London, UK, Jan. 2001. Springer.

- **Index structures for NN-search**

  - R.A. Finkel and J.L. Bentley. **Quad-trees: A data structure for retrieval on composite keys**. ACTA Informatica, 4(1):1-9, 1974
  - J.T. Robinson. **The k-d-b-tree: A search structure for large multidimensional dynamic indexes**. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 10-18, 1981.
  - J. Nievergelt, H. Hinterberger, and K.C. Sevcik. **The grid file: An adaptable symmetric multikey file structure**. ACM Transactions on Database Systems, 9(1):38-71, March 1984.
  - J. A. Orenstein and T. H. Merrett. **A Class of Data Structures for Associative Searching**. In Proceedings of the ACM Symposium on Principles of Database Systems (PODS), pages 181--190, Waterloo, Ontario, Canada, Apr. 1984. ACM.
  - A.Guttman. **R-Trees: A dynamic index structure for spatial searching**. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 47-57, Boston, MA, June 1984.
  - T. Sellis, N. Roussopoulos, and C. Faloustos. **The R+-tree: A dynamic index for multi-dimensional objects**. In Proceedings of the International Conference on Very Large Databases (VLDB), pages 507-518, Brighton, England, 1987.
  - N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. **The R*-tree: An efficient and robust access method for points and rectangles**. In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, pages 322-331, Atlantic City, NJ, 23-25 May 1990.
  - H. V. Jagadish. **Spatial Search with Polyhedra**. In Proceedings of the International Conference on Data Engineering (ICDE), pages 311--319, Los Angeles, California, USA, Feb. 1990. IEEE Computer Society.
  - K.-I. Lin, H.V. Jagadish, and C. Faloutsos. **The TV-tree: An index structure for high-dimensional data**. The VLDB Journal: The International Journal on Very Large Data Bases, 3(4):517-549, October 1994.
  - Tzi-cker Chiueh. **Content-Based Image Indexing (vp-Tree)**. In Proceedings of the Twentieth International Conference on Very Large Databases, pages 582--593, Santiago, Chile, 1994.
  - S. Berchtold, D.A. Keim, and H.-P. Kriegel. **The X-tree: An index structure for high-dimensional data**. In Proceedings of the International Conference on Very Large Databases (VLDB), pages 28-39, 1996.
  - David A. White, Ramesh Jain: **Similarity Indexing with the SS-tree.** ICDE 1996: 516-523

- **Index structures for NN-search**

  - R. Kurniawati, J. S. Jin, and J. Shepherd. **SS+-Tree: An Improved Index Structure for Similarity Searches in a High-Dimensional Feature Space**. In Storage and Retrieval for Image and Video Databases (SPIE), volume 3022 of SPIE Proceedings, pages 110--120, San Jose, CA, USA, Feb. 1997.
  - N. Katayama and S. Satoh. **The SR-tree: An index structure for high-dimensional nearest neighbor queries**. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 369-380, Tucson, Arizon USA, 1997.
  - P. Ciaccia, M. Patella, and P. Zezula. **M-tree: An efficient access method for similarity search in metric spaces**. In Proceedings of the International Conference on Very Large Databases (VLDB), Greece, 1997.
  - C. Böhm and H.-P. Kriegel. **Dynamically Optimizing High-Dimensional Index Structures**. In Proceedings of the International Conference on Extending Database Technology, volume 1777 of Lecture Notes in Computer Science, pages 36--50, Konstanz, Germany, Mar. 2000. Springer-Verlag.
  - J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. **Generalized Search Trees for Database Systems (GiST)**. In Proceedings of the International Conference on Very Large Databases (VLDB), pages 562-573, Zurich, Switzerland, Sept. 1995. Morgan Kaufmann.
  - Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. **The pyramid-technique: Towards breaking the curse of dimensionality**. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 142-153, Seattle, WA, USA, 1998
  - J. Goldstein and R. Ramakrishnan. **Contrast Plots and P-Sphere Trees: Space vs. Time in NN Searches**. In Proceedings of the International Conference on Very Large Databases (VLDB), pages 429--440, Cairo, Egypt, Sept. 2000. Morgan Kaufmann.
  - **R-Tree Visualization**, http://www.dbnet.ece.ntua.gr/~mario/rtree/.

- **VA-File**
  - Roger Weber, Hans-J. Schek and Stephen Blott. **A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces**. In Proceedings of the International Conference on Very Large Databases (VLDB), New York, USA, August 1998.
  - Roger Weber, Klemens Böhm. **Trading Quality for Time with Nearest-Neighbor Search**. In VII. Conference on Extending Database Technology (EDBT'2000), Konstanz, Germany, March 27-31 2000.
  - Roger Weber, Klemens Böhm, and Hans-J. Schek. **Interactive-Time Similarity Search for Large Image Collections Using Parallel VA-Files**. In 16th International Conference on Data Engineering (ICDE'2000), San Diego, CA, USA, February 29 - March 3, 2000.
  - Roger Weber, **Similarity Search in High-Dimensional Vector Spaces**, PhD. Thesis, ETH Zurich, 1999.

- **Complex Queries**
  - P. Ciaccia, M. Patella, and P. Zezula. Processing Complex Similarity Queries with Distance-Based Access Methods. In Proceedings of the International Conference on Extending Database Technology, volume 1377 of Lecture Notes in Computer Science, pages 9-23, Valencia, Spain, Mar. 1998. Springer.
  - R. Fagin. Combining Fuzzy Information from Multiple Systems. In Proceedings of the ACM Symposium on Principles of Database Systems (PODS), pages 216-226, Montreal, Canada, June 1996. ACM Press.
  - Surajit Chaudhuri, Luis Gravano. **Optimizing Queries over Multimedia Repositories**, SIGMOD Conf. 1996, pages 91-102.
  - Surajit Chaudhuri,Kyuseok Shim. **Optimization of Queries with User-defined Predicates**, VLDB 1996, pages 87-98.
  - M. Mlivoncic, K. Böhm, H.-J. Schek, R. Weber: **Fast Evaluation Techniques for Complex Similarity Queries**. 27th Int. Conf. on Very Large Databases (VLDB), Roma, Italy, September 2001.