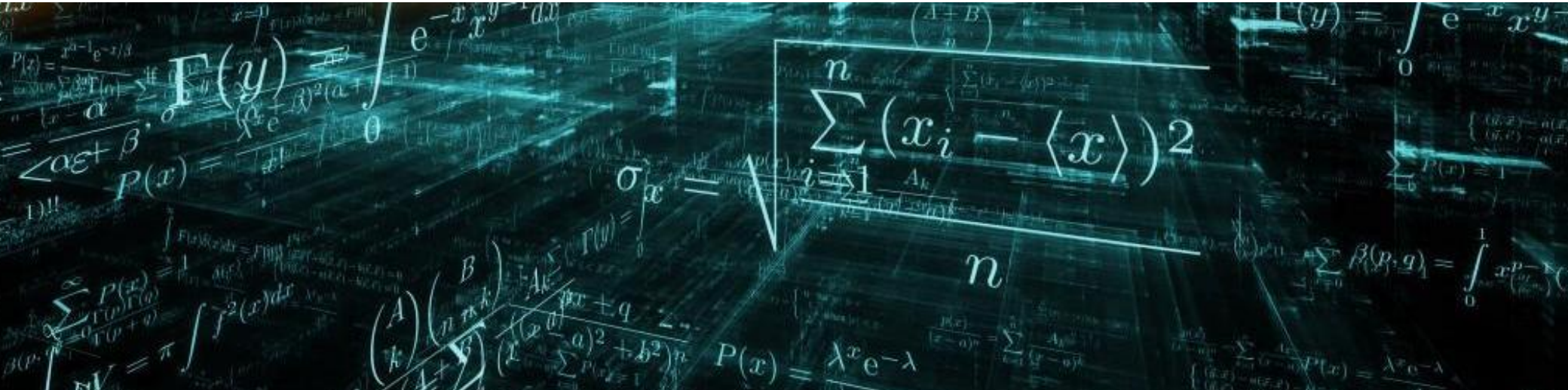
Universität Basel

Computer Science / 15731-01 / 2025

Multimedia Retrieval

Chapter 7: Vector Search

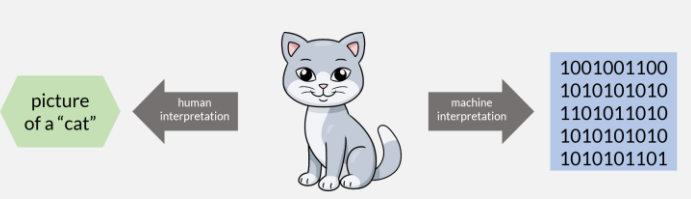
Dr. Roger Weber, roger.weber@gmail.com



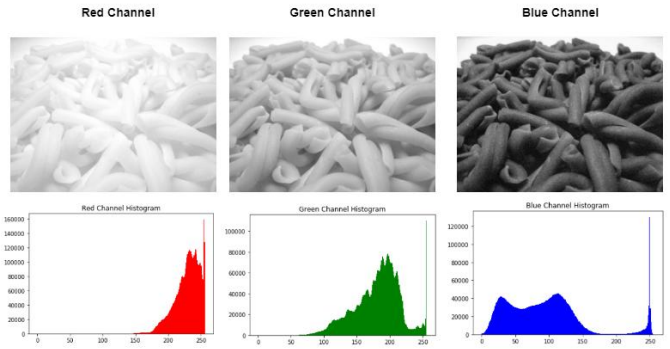
7.1 Introduction

- The story of vector search does not begin with algorithms or system architectures but with a basic problem that has challenged information retrieval systems since the beginning: the semantic gap. This gap, the mismatch between low-level computational features and high-level human understanding, has driven decades of innovation in how we search for, retrieve, and make sense of digital information.
- In the 1990s, as digital image collections grew rapidly, researchers ran into a key limit of traditional retrieval systems. A user searching for "vacation photos" expected the system to grasp the idea: beaches, sunsets, families together. Keyword-based metadata could only match exact text labels. Images without proper tags stayed hidden, and manual tagging was too slow and subjective to scale.
- Content-based Image Retrieval (CBIR) was the first systematic effort to close this gap. IBM's QBIC system in 1995 introduced feature extraction, turning images into vectors that describe color, texture, and shape, and allowed comparisons based on visual traits rather than metadata alone. However, these low-level features captured only surface properties. For example, a blue ocean and a blue sky can have similar color histograms while meaning very different things. The semantic gap remained: machines could detect pixel patterns but could not grasp meaning.
- This challenge went well beyond images. Text retrieval had similar limits. Traditional inverted indexes and TF-IDF systems worked for exact keyword matches but struggled with synonyms, paraphrases, and conceptual queries. For example, searching for “heart disease” did not find documents about “cardiology treatment”. The systems could not capture semantic relationships, which reduced their usefulness across all modalities: text, images, audio, and video.

Semantic Gap

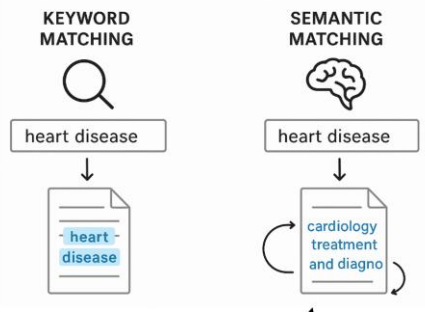


Content-based Image Retrieval



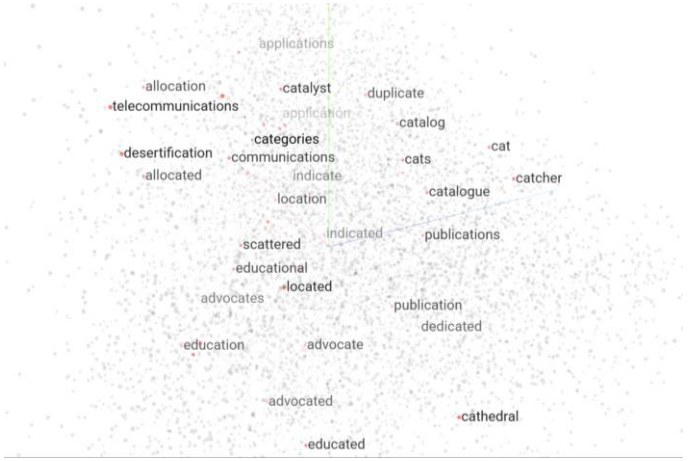
source: https://theailearner.com/2019/02/09/understanding-image-histograms

Key-word vs. Semantic Matching



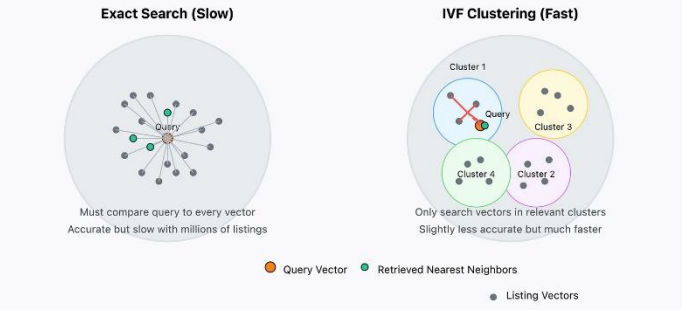
- The breakthrough came with neural embedding models had matured enough to put meaning into numbers. In 2013, Word2vec showed that semantic relations can emerge from vector math. In 2018, BERT introduced contextualized embeddings that capture how a word's meaning shifts with its surrounding text. In 2021, CLIP created multimodal embeddings that place text and images in a shared semantic space, enabling zero-shot cross-modal search. These models learned representations that reflect human ideas of similarity and meaning.
- Vector search became the infrastructure that lets embeddings work at scale. It turns queries and documents into dense vectors in high-dimensional space, so systems can judge meaning by how close vectors are to each other. A query vector for "vacation photos" will sit near image vectors for beaches and families, even when the images have poor tags. Nearest neighbor search became a key part of the retrieval pipeline.
- As data volumes grew exponentially, a new barrier appeared: scale. Brute force search for the most similar vectors among millions or billions of candidates required prohibitive computing resources. Traditional spatial indexes like R-trees handle a few dimensions but fail under the curse of dimensionality. As vector dimensions increase, distances between points converge and the idea of a nearest neighbor loses meaning.
- Exact nearest neighbor (NN) search looks ideal in theory, but in practice it provides little benefit for the applications that use it. Embedding-based systems care about finding relevant results, not exact geometric closeness in the embedding space. Approximate nearest neighbor (ANN) search optimizes for the metric that actually matters: how often the retrieved results meet the downstream task. A document or product that ranks third in the embedding space is often still relevant to a user's query. Empirically, ANN algorithms achieve over 95 percent of the recall of exact search, while running hundreds of times faster and scaling to billions of items.

Search in Embedding Space



source: https://projector.tensorflow.org

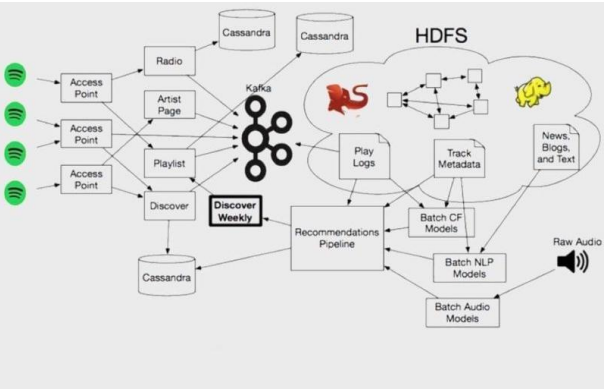
Trade-off with Approximate Nearest Neighbour Search



source: https://stacksweep.substack.com/p/smarter-search-at-airbnb-how-embedding

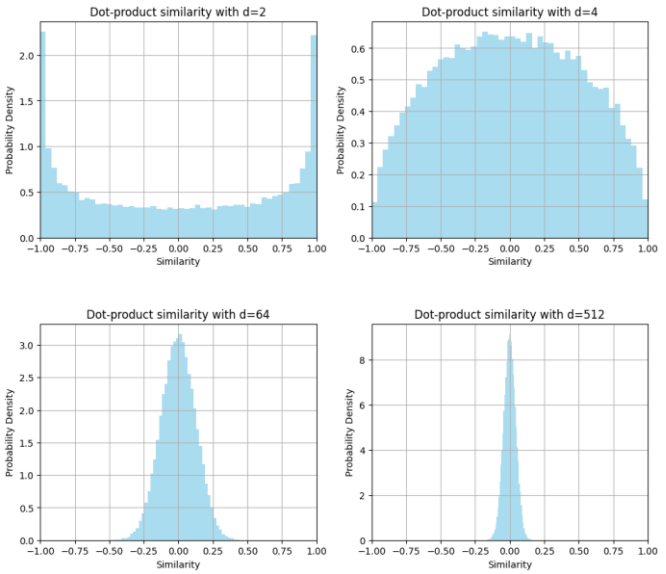
- Today, vector search supports a wide range of applications. Spotify recommends music using embeddings of audio features. Financial firms detect fraud by finding unusual transaction patterns in high-dimensional behavioral spaces. Healthcare systems match patients with similar symptom profiles to guide treatment decisions. Autonomous vehicles process sensor streams as vectors to navigate in real time.
- The infrastructure evolved in parallel. Specialized libraries such as FAISS (2017) offered GPU-accelerated similarity search with sophisticated compression techniques. Purpose-built vector databases like Milvus and Pinecone were optimized for high throughput and low latency retrieval. Traditional systems added vector capabilities, for example pgvector for PostgreSQL and Elasticsearch's k-NN search, allowing organizations to adopt vector search without replacing their entire infrastructure.
- Even though vector search is mature, it still faces technical hurdles. At very large scale, the curse of dimensionality makes distance measures less reliable, so work focuses on quantization and dimensionality reduction. Another challenge is interpretability. Dense embeddings capture meaning as a whole but are hard for humans to inspect, while sparse features as with BM25 are easier to understand but miss subtle meaning. A growing consensus favors hybrid retrieval, which blends dense and sparse signals using ranking fusion methods such as Reciprocal Rank Fusion (RRF).
- The semantic gap that once separated human thought from machine retrieval is steadily closing. A new information paradigm is emerging: every kind of data, from legal contracts and jazz riffs to medical scans, can be represented, compared, and retrieved by meaning. Vector search is no longer just a technology; it is the foundation of semantic search, where similarity is measured by understanding rather than by words or symbols.

Spotify Recommendation Approach



source: https://www.linkedin.com/posts/grusiya_systemdesign-spotify

Curse of Dimensionality



Category	High Adoption / High Impact	Emerging Adoption / High Impact
High Business & Technical Impact	• **Semantic Search and Information Retrieval:** Enables users to find relevant information across large document collections by understanding the meaning behind queries and content. • **Recommendation Systems:** Delivers personalized product or content suggestions by analyzing user behavior, preferences, and item similarities using vector-based embeddings. • **Retrieval-Augmented Generation (RAG):** Combines language models with external knowledge sources to provide accurate, context-aware answers and support enterprise AI applications. • **Fraud and Anomaly Detection:** Identifies unusual patterns in transactions or behavior using vector similarity to detect fraud and emerging security threats efficiently.	• **Healthcare and Drug Discovery:** Uses patient data, medical records, and molecular embeddings to identify similar cases, predict outcomes, and accelerate pharmaceutical research. • **Multimodal Search:** Enables searching across text, images, audio, and video by embedding different data types in a shared vector space for retrieval.
Moderate to Niche Impact	• **Image and Video Similarity Search:** Transforms images and video into vector representations to enable searches based on visual similarity, improving discovery in e-commerce and media libraries. • **Audio and Music Search:** Converts audio into embeddings capturing rhythm, melody, and acoustic features to support music recommendations, sound identification, and audio retrieval applications.	• **Financial Risk Analysis and Compliance:** Uses vector representations of market data, news sentiment, and regulatory documents to identify investment opportunities and ensure compliance efficiently. • **Geospatial and Autonomous Systems:** Processes location and sensor data as vectors to optimize navigation, route planning, urban analysis, and decision-making in specialized domains.

7.2 Distance and Similarity Measures

- Let us first examine embeddings. Each document D_i produces an M-dimensional embedding vector $\boldsymbol{d}_i$. Similarly, a query can be represented as an embedding vector $\boldsymbol{q}$. Unlike classical feature representations for images, embeddings can have both positive and negative values, whereas classical retrieval methods use mostly positive values.
  - We can measure similarity between a document and a query using either the dot product or cosine similarity. In both cases, higher values indicate a better match. The figures on the right show dot-product similarity and cosine similarity in a two-dimensional example.

$$sim_{dot}(Q, D_i) = \boldsymbol{q} \cdot \boldsymbol{d}_i$$

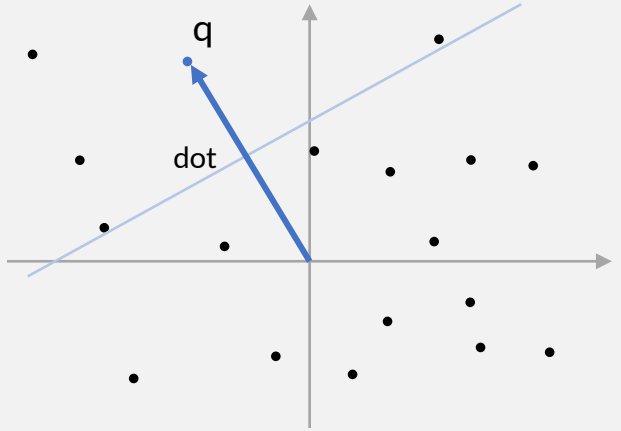
$$sim_{cos}(Q, D_i) = \frac{\boldsymbol{q} \cdot \boldsymbol{d}_i}{\|\boldsymbol{q}\| \cdot \|\boldsymbol{d}_i\|}$$

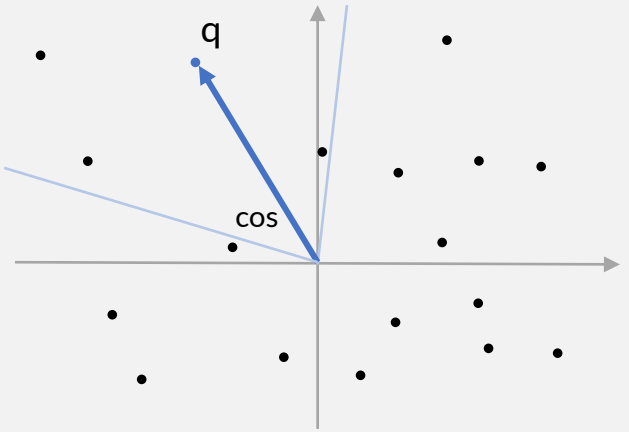
  - For the dot product, a plane perpendicular to the query vector separates the top documents, here the top three, from the others. This plane can include documents in different quadrants, but generally not in the quadrant opposite the query.
  - Cosine similarity defines a cone around the query vector that identifies the best-matching documents. This cone can include documents in other quadrants, but generally not in the quadrant opposite the query.
  - **Best Match Problem:** Given document vectors $\boldsymbol{d}_i$ for $1 <= i <= N$ and a query vector $\boldsymbol{q}$, find the document vector $\boldsymbol{d}^*$ that maximizes a similarity measure, such as the dot product or cosine similarity.

$$\boldsymbol{d}^*_{dot} = \underset{1 \le i \le N}{\operatorname{argmax}}\, \boldsymbol{q} \cdot \boldsymbol{d}_i$$

$$\boldsymbol{d}^*_{cos} = \underset{1 \le i \le N}{\operatorname{argmax}}\, \frac{\boldsymbol{q} \cdot \boldsymbol{d}_i}{\|\boldsymbol{q}\| \cdot \|\boldsymbol{d}_i\|}$$

  - More generally, we can look for the top k matches, denoted d^*_k, which have the highest similarity scores.





- To simplify our discussion, we can merge the two methods into one. When using the cosine measure, we first normalize each vector by its length and store these normalized document vectors in the index. This lets us use the simpler dot product. Normalization maps every vector onto the unit sphere centered at the origin.

$$\widehat{\boldsymbol{d}} = \frac{\boldsymbol{d}_i}{\|\boldsymbol{d}_i\|} \qquad \widehat{\boldsymbol{q}} = \frac{\boldsymbol{q}}{\|\boldsymbol{q}\|} \qquad sim_{cos}(Q, D_i) = \frac{\boldsymbol{q} \cdot \boldsymbol{d}_i}{\|\boldsymbol{q}\| \cdot \|\boldsymbol{d}_i\|} = \widehat{\boldsymbol{q}} \cdot \widehat{\boldsymbol{d}} = sim_{dot}(\widehat{Q}, \widehat{D_i})$$

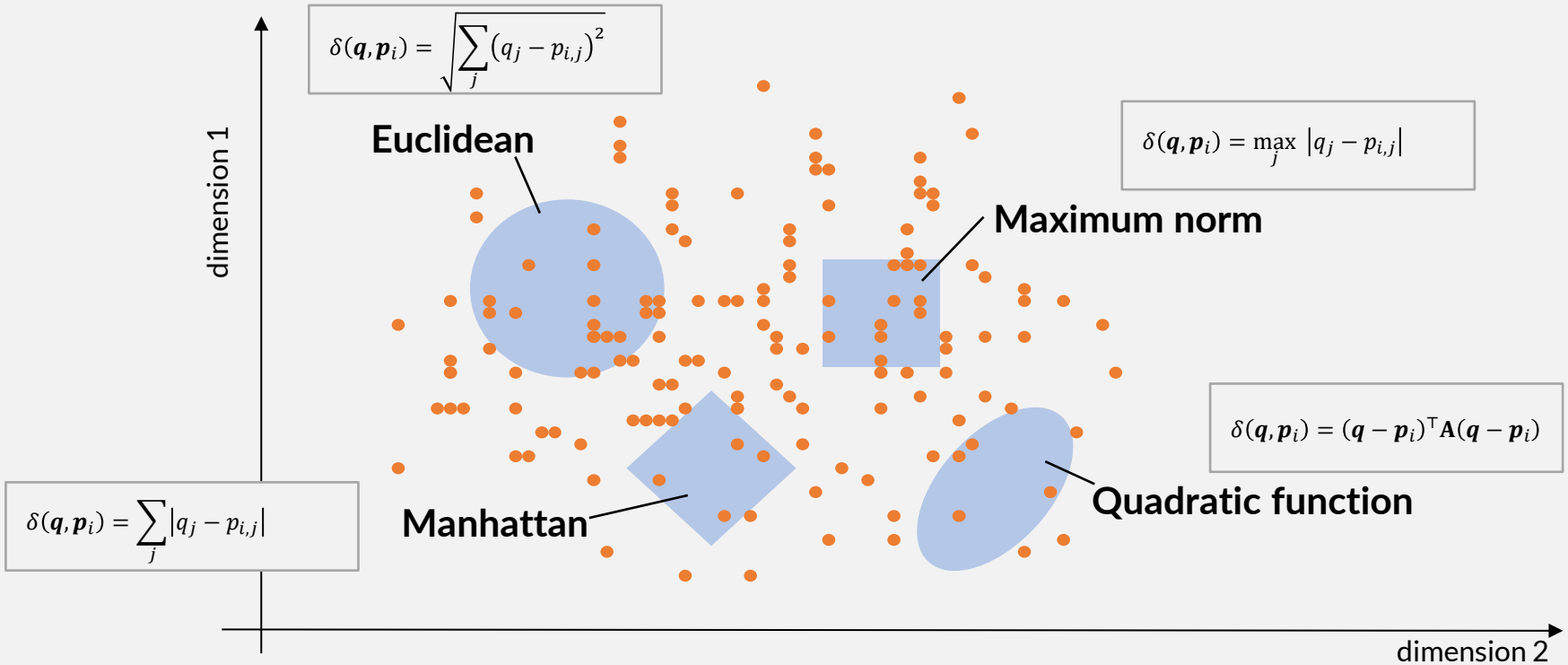
- Another point about dot-product embeddings is that the similarity increases when query and document vector components share the same sign and decreases when their signs differ.

$$sim_{dot}(Q, D_i) = \boldsymbol{q} \cdot \boldsymbol{d}_i = \sum_{k=1}^{D} q_k \cdot d_{i,k} \qquad q_k \cdot d_{i,k} \begin{cases} \geq 0 & \text{if } sign(q_k) = sign(d_{i,k}) \\ \leq 0 & \text{if } sign(q_k) \neq sign(d_{i,k}) \end{cases}$$

This does not mean we can ignore documents with components that have opposite signs from the query. A document can still score highly if many of its components align with the query. With inverted files, sparsity allowed us to ignore many components of the query. This is not true for embeddings. Even a small query component q_k can contribute significantly to the total similarity if the corresponding document component $d_{i,k}$ is large.

- Content-based retrieval, or CBIR, extracts feature representations directly from raw data to enable similarity-based searches. Examples of such features include color histograms for capturing color distributions in images, texture descriptors such as Gabor filters, and shape or edge descriptors like SIFT or HOG. For audio, common features include Mel-frequency cepstral coefficients (MFCCs), chroma features, and rhythm or tempo. For video, features often combine frame-level image descriptors, motion vectors, optical flow, and temporal patterns to capture both visual and dynamic content. Each feature is carefully designed with a specific distance measure in mind, ensuring that the feature and its distance metric together effectively represent the intended notion of similarity.
  - Distance measures behave differently from similarity measures. A smaller distance indicates higher similarity, while a larger distance indicates lower similarity. In CBIR, the search is usually for the nearest neighbor, which is the feature vector closest to the query vector for the chosen distance metric.
  - Interpreting distance values is often more difficult than interpreting normalized similarity scores between 0 and 1. For example, a distance of 32.8 does not immediately indicate whether two objects are very similar or quite different. Similarity measures, on the other hand, provide an intuitive sense of closeness, where values close to 1 clearly indicate stronger resemblance.

- In content-based retrieval, several distance metrics are commonly used:
  - **Euclidean distance** calculates the straight-line distance between two vectors in high-dimensional space. It is widely used for continuous features and gives an intuitive geometric measure of similarity.
  - **Manhattan distance** sums the absolute differences of each vector component. This metric is useful when features represent independent dimensions or when large individual differences should have a stronger effect.
  - **Maximum distance** considers only the largest absolute difference between corresponding vector components. It is suitable where the largest deviation dominates similarity judgments and outliers should be emphasized.
  - **Quadratic distance**, sometimes called weighted Euclidean distance, accounts for correlations between vector components. It applies a weighting matrix to scale or rotate the space so that differences along certain dimensions contribute more to the overall distance. This metric is particularly useful when feature dimensions have different variances or are not independent.



- **Normalization of vectors**: When creating feature vectors from different low-level components, it is important to ensure that the value ranges are comparable.
  - For example, consider a two-dimensional vector where dimension d_1 ranges from 0 to 1 and dimension d_2 ranges from 0 to 1000. Because d_2 has a much larger range, distance measures like Euclidean distance are dominated by differences in d_2, making d_1 contribute very little to similarity calculations.
  - A simple and effective approach is Gaussian normalization. For each dimension, calculate the mean μ_j and standard deviation σ_j, either from the full dataset or a large sample. Exact values are not required, and they can be kept constant over time. Each vector $\boldsymbol{v}$ is then transformed into its normalized version $\hat{\boldsymbol{v}}$ by subtracting the mean and dividing by the standard deviation. Applying the same transformation to both queries and media items removes the mean from distance calculations and scales each dimension by its standard deviation.

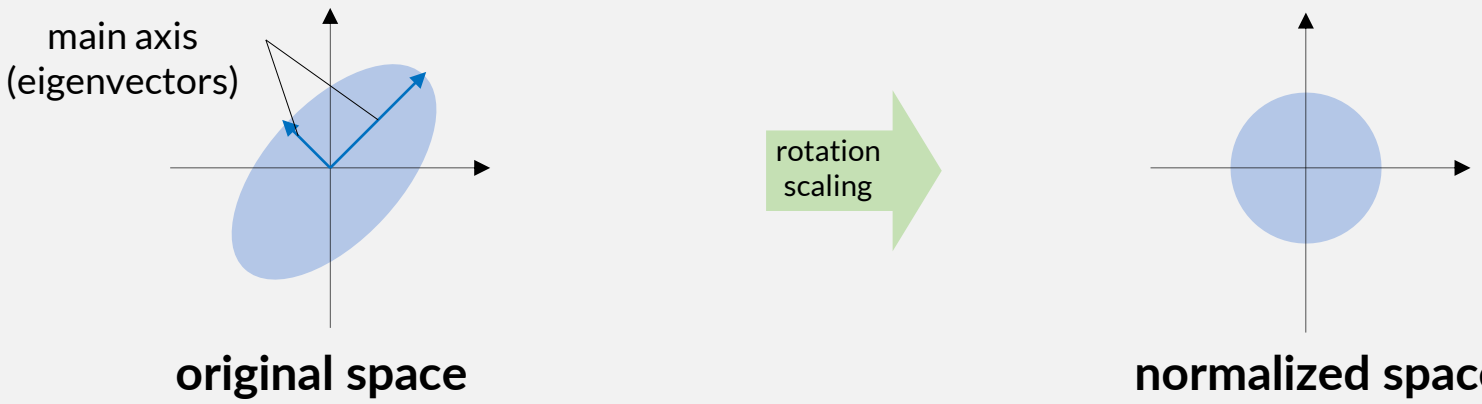
$$\hat{v}_j = \frac{v_j - \mu_j}{\sigma_j}$$

  - From another perspective, Gaussian normalization converts the distance measure into a weighted distance, for example in Manhattan distance, with each dimension scaled according to its variance. This approach allows the original feature vectors to be preserved while incrementally computing weights w_j as new vectors are added. Useful parameters for the index include sums of component values and sums of squared component values, which can be used to compute standard deviations and weights.

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

  - If vector components are not independent, dimensionality reduction methods such as PCA or SVD can produce more compact representations. These methods transform high-dimensional vectors into lower dimensions while retaining essential information. The transformation can be stored with the index for use with new vectors and queries. Reapplying the transformation periodically ensures it remains accurate as the dataset evolves.

- Quadratic distance functions are used to account for dependencies between vector components. For example, in color histograms, each bin represents a reference color, and its value counts the number of pixels closest to that color. The bins are not independent. For instance, the bin for red is related to the bin for orange because red and orange are perceptually similar, while red and blue are not.
  - To create a quadratic distance function, we represent the relationships between components using a $d \times d$ matrix $\mathbf{A}$. This matrix must be positive semi-definite to ensure all vector differences produce non-negative distances, meaning $\boldsymbol{x}^\top \mathbf{A} \boldsymbol{x} \geq 0$ for any vector $\boldsymbol{x}$. Linear algebra tells us that such a matrix defines a hyper-ellipse, with eigenvectors indicating the main axes and eigenvalues determining scaling along those axes.
  - Using eigenvalue decomposition, we can rotate and scale the original space, creating a normalized space with the same dimensions. In this transformed space, the quadratic distance function becomes equivalent to a squared Euclidean distance. For a given matrix $\mathbf{A}$, we can rotate all data vectors and the query into this normalized space and then measure similarity using standard Euclidean distance.
  - This transformation makes the search much more efficient because it eliminates the need for expensive vector-matrix-vector multiplications at search time. Once the data and query vectors are rotated and scaled, similarity calculations only require simple Euclidean distance computations, significantly reducing computational cost.



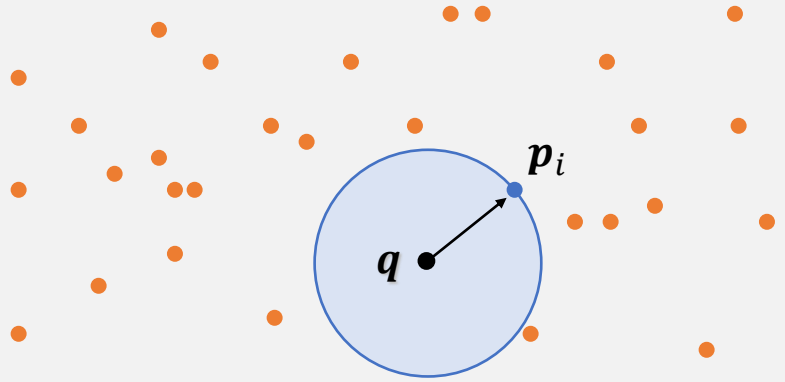
- In a similarity search, finding the most similar vector to a query vector means seeking the vector $\boldsymbol{p}_i$ that maximizes similarity for embeddings or minimizes the distance for feature vectors. With embeddings, we refer to $\boldsymbol{p}_i$ as the **most similar** vector for a query $\boldsymbol{q}$ under a given similarity measure. For feature vectors, we denote $\boldsymbol{p}_i$ as the **nearest neighbor** to $\boldsymbol{q}$ in the database (=$NN(\boldsymbol{q})$) under a given distance:

$$MS(\boldsymbol{q}) = \underset{\boldsymbol{p}_i \in index}{\operatorname{argmax}}\, sim(\boldsymbol{p}_i, \boldsymbol{q}) \qquad NN(\boldsymbol{q}) = \underset{\boldsymbol{p}_i \in index}{\operatorname{argmin}}\, dist(\boldsymbol{p}_i, \boldsymbol{q})$$

  - Converting a "most similar" search problem into a "nearest neighbor" search problem is straightforward.
- In this chapter, the focus is on nearest neighbor search, with additional guidance on optimizing searches for similarity measures when needed. The main emphasis is on Euclidean and Manhattan distances, along with the dot product for similarity calculations. We have also explained how the cosine measure can be converted into a dot product.
  - The chapter explores nearest neighbor search using a selected set of index methods. This topic is currently a major focus of research, especially for semantic search with embeddings. We begin with techniques suitable for lower-dimensional spaces and then discuss challenges that arise from the "curse of dimensionality" in higher dimensions. In high-dimensional spaces, the most effective approaches use a relaxed, approximate search strategies to locate nearby neighbors rather than exact nearest neighbors.

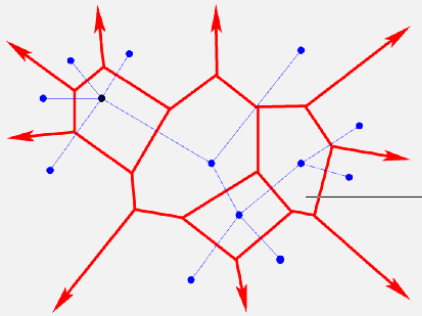
Nearest Neighbor Problem:

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



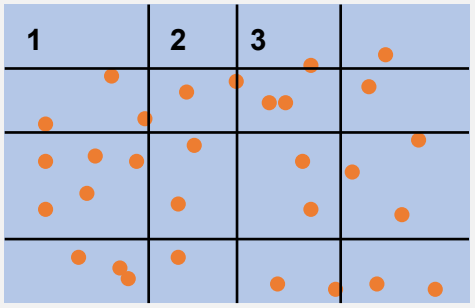
7.3 Low-dimensional Search Structures

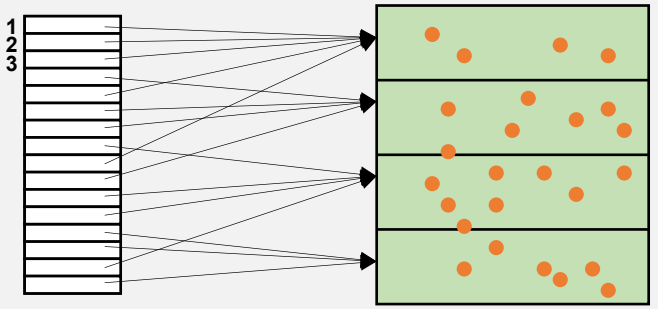
- A **Voronoi diagram** is a geometric structure that divides a space into regions based on the closest distance to a set of seed points. Voronoi diagrams are particularly useful in various spatial applications, such as geographical information systems (GIS), computational geometry, nearest neighbor search, and are widely used in algorithms and data structures for spatial data analysis and optimization.
  - Given a set of seed points, a Voronoi diagram partitions space so each region contains all locations closest to one seed point. Each region then represents precomputed results of nearest neighbor searches. Instead of storing data points directly, we compute the Voronoi diagram and index its regions. To find the nearest neighbor for a query point, we find which region that contains the query point and return that region's seed point as the answer. Voronoi diagrams can be built for different distance measures, not only for Euclidean distances shown below.
  - Two main challenges are computing the Voronoi diagram, especially in higher dimensions, and storing the cells so point-containment queries are fast. In the example below, some regions are bounded by six or more lines, which raises storage needs exponentially with dimensionality. To reduce storage costs, we can approximate a Voronoi cell by its minimum bounding rectangle. That may require checking several regions instead of just one to find the nearest point. The extra work is small in low dimensions but can grow quickly in higher dimensions, as we will see.
  - Voronoi diagrams function somewhat like inverted files in text retrieval, providing answers to all potential queries similar to inverted files for single-term queries. However, it is not an ideal data structure due to complex insertion and removal operations, and limited scalability in higher dimensions.



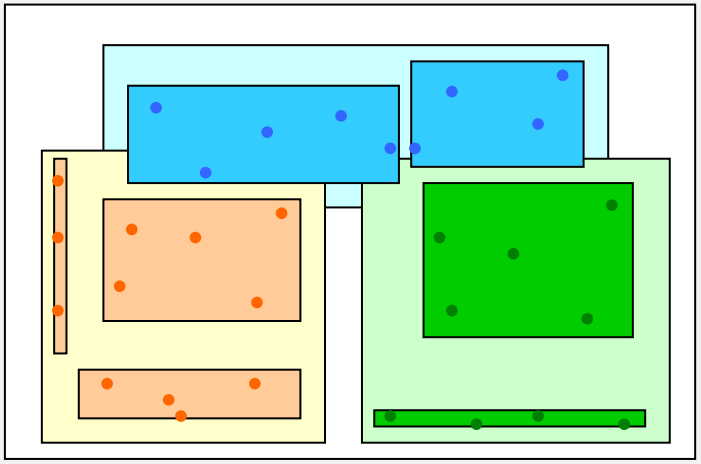
Voronoi Cell: all points in the cell are closest to blue center point

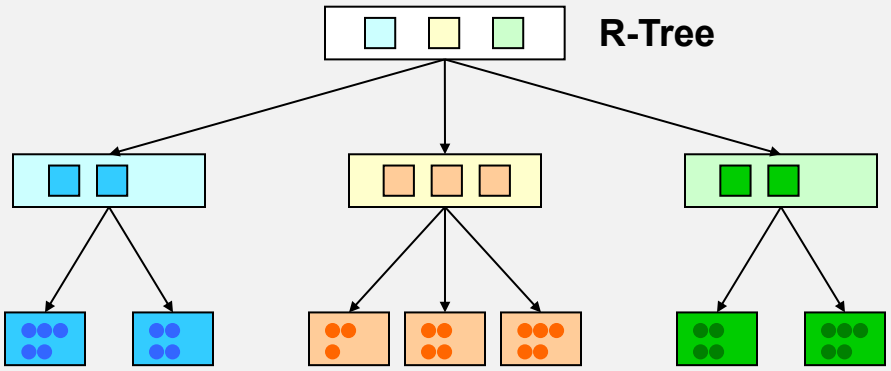
- A **gridfile** is a spatial data structure used to organize and index multidimensional data efficiently. It works by dividing the multidimensional space into a grid of cells, where each cell is represented by a bucket:
  - **Grid Partitioning**: the initial step is to divide the multidimensional space into a grid, where the grid cells create a rectangular layout. The gridlines may not be equally spaced, but the division is designed to ensure that the resulting pages on disk hold roughly the same number of data points.
  - **Dictionary:** grid cells are organized in a dictionary, which associates a cell with a disk page holding the cell's data points. Disk pages can consolidate data from multiple cells to optimize storage when many cells remain empty due to data distribution.
  - **Insertions:** when adding new data points to the gridfile, disk pages may become full. In this scenario, two new pages are created, and cells are evenly distributed between them. If a cell grows too large for a disk page, the gridfile introduces a new partition line along a chosen dimension, creating new cells. This update affects the dictionary but only necessitates redistributing the data points from the overflowing cell to the new disk pages.
  - **Query Processing**: to perform a range query, the gridfile divides the query space into the corresponding grid cells and retrieves data points from these cells. The advantage of this approach is that it significantly reduces the search space, making queries faster. For nearest neighbor search, we will explore a generic algorithm after the next method that is applicable to the gridfile as well.
  - Gridfiles are particularly useful for efficient spatial data indexing in databases. They are commonly employed in geographic information systems (GIS), image databases, and various applications where multidimensional data needs to be quickly retrieved based on spatial attributes. Gridfiles are more suitable for moderate to high-dimensional data compared to Voronoi diagrams, as they offer better scalability and performance in higher dimensions. In lower dimensions, they offer quick responses with a few data page reads.



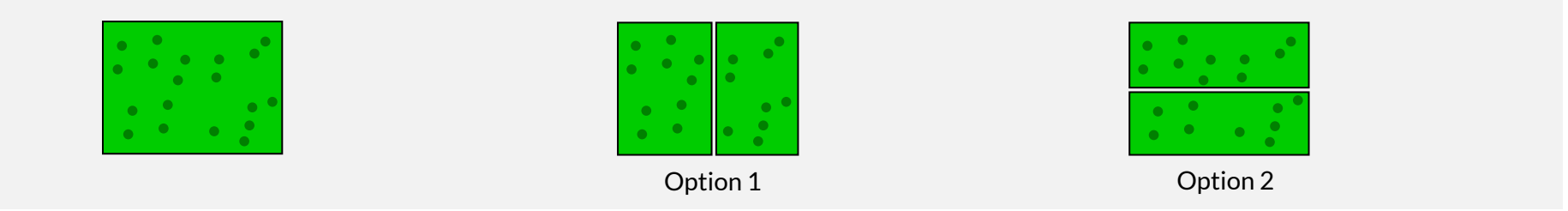


- An **R-tree** is a tree-based spatial data structure used for indexing and searching multidimensional data in a space, particularly in geographic information systems (GIS) and database systems. It is designed to efficiently store and query spatial objects like points, rectangles, and polygons similar to a B-tree in conventional databases:
  - **Structure:** an R-tree is a balanced tree structure where each node represents a bounding rectangle that encloses a set of data points or other bounding rectangles. The root of the tree encompasses all the objects.
  - **Hierarchical Organization**: the tree's hierarchical organization means that as you traverse the tree from the root to the leaves, you progressively narrow down the search space. Data points typically reside in leave nodes only.
  - **Insertion**: when you insert a datapoint into an R-tree, the structure determines where to place it within the tree. It selects a leaf that can accommodate the new object without violating the tree's balance and spatial structure.
  - **Splitting and Reorganization**: if a node becomes too full, it may be split into two, and the parent node is updated. This may lead to consecutive splits up to the root if also parent nodes become full. If the root node overflows, it is split into two parts and a new root is created. R-trees are designed to keep their structure balanced. This means that each node contains a roughly equal number of data points or child nodes, which ensures that the tree remains efficient for search operations.
  - **Query Processing**: to perform a query, such as a range or nearest neighbor search, you start at the root and traverse the tree, visiting only those nodes that overlap or are close to the query region. This significantly reduces the number of objects to be considered, making queries much faster than a brute-force search.

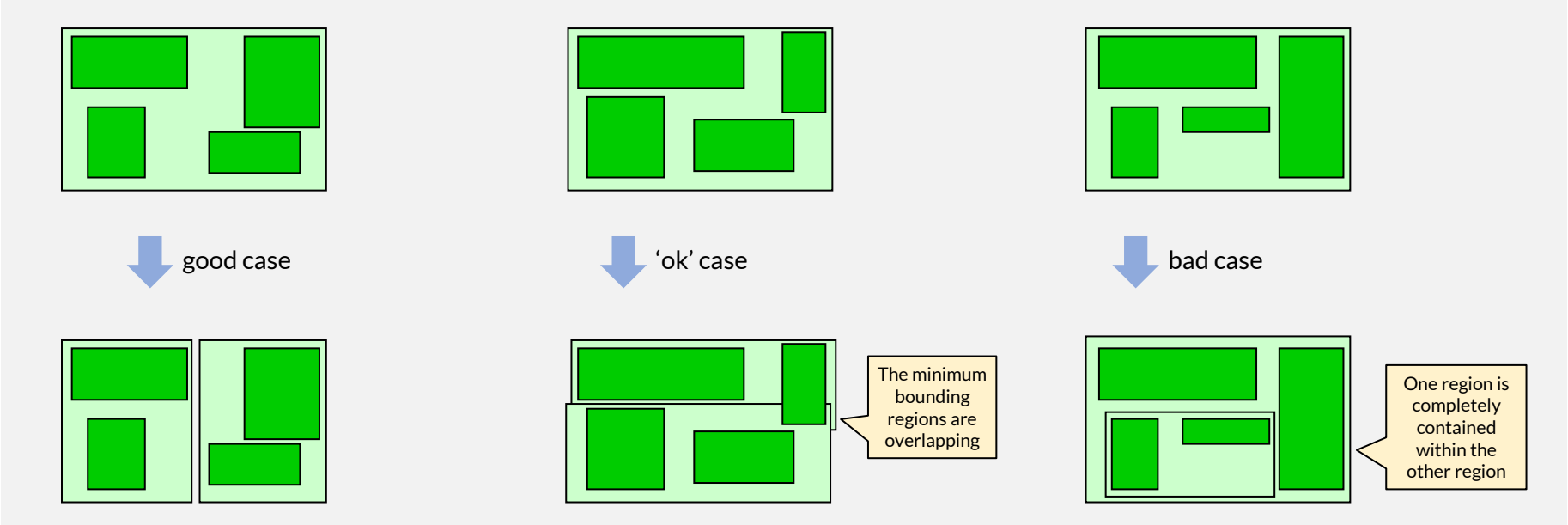




- Splitting nodes in an R-tree can lead to inefficiencies:
  - Splitting leaf nodes is straightforward: data points are divided along a dimension using the median value. This split guarantees that the resulting leaf nodes cover non-overlapping areas, ensuring efficient search operations by preventing overlaps that might require visiting multiple nodes during insertion and searches.

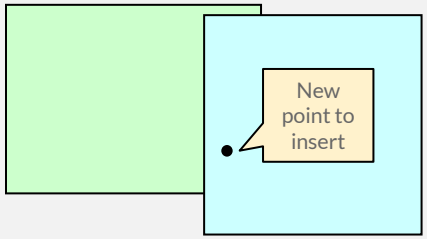


  - Splitting inner nodes can lead to overlaps, as it is often not feasible to consistently separate the minimum bounding rectangles of child nodes into 2 non-overlapping areas, as illustrated in the examples below:

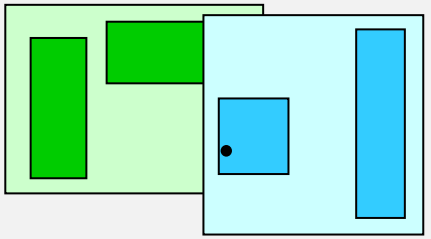


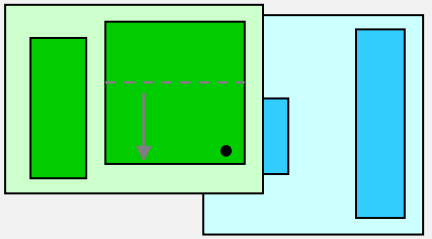
  - Certain R-tree variants minimize overlaps by re-inserting data points from leaves with substantial overlap. Another approach involves bottom-up reconstruction of the R-tree to eliminate leaf-level overlaps, which subsequently prevents overlaps in inner nodes.

- **Overlaps are a problem:** Overlaps force searches to examine multiple child nodes that share the same regions. Furthermore, overlaps make insertions unpredictable and can cause more overlaps if the insertion path is poor, because leaf nodes may grow and start overlapping with neighboring leaves.



Which path to follow?


good case: follow blue path

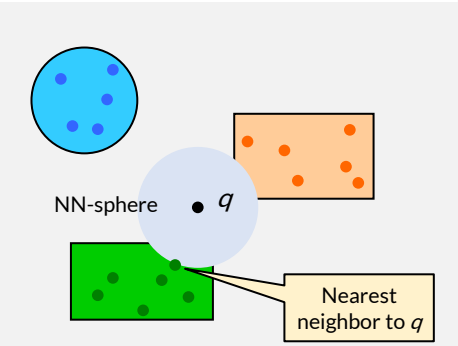

bad case: follow green path

- **R-tree extensions:** Over time, several extensions and optimizations for R-trees have been introduced. Key optimization aspects include:
  - Minimum bounding region shapes (e.g., rectangles, spheres, or combinations)
  - Splitting methods to reduce leaf node overlaps
  - Adjusting node sizes, such as increasing page size if splitting is unbeneficial
  - Metric Trees, which rely on object metrics (e.g., edit distance) rather than a data space metric (like Euclidean)
  - Notable examples include 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 more.

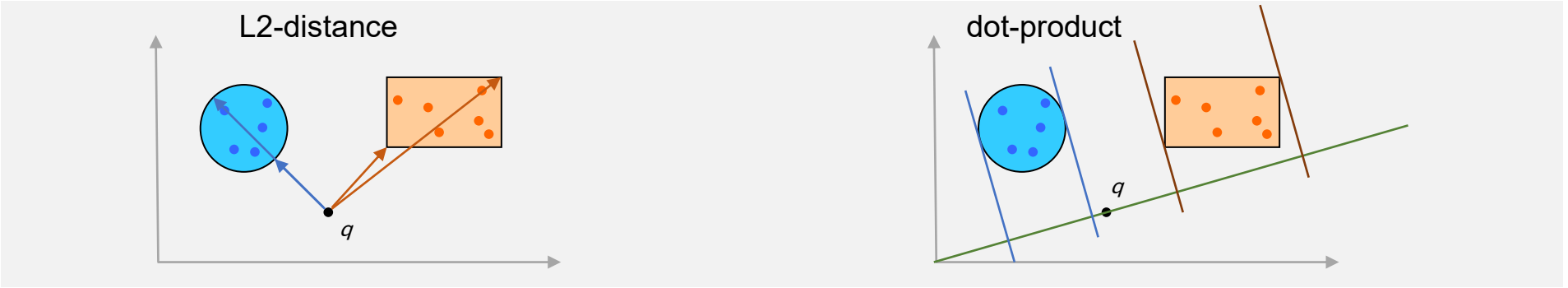
- **Hjaltson and Samet** proposed an optimal search algorithm for identifying the nearest neighbor within hierarchical structures. "Optimal" in this context implies that the algorithm minimizes the number of visited nodes, ensuring the correctness of the nearest neighbor found (which cannot be achieved with fewer visits).
  - The algorithm employs a priority queue for nodes and points. Priority is determined by the distance between the query point and data point or the minimal bounding region (MBR). The queue is sorted by distances in ascending order. The algorithm operates as follows for a given query object $\boldsymbol{q}$:

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

  - The algorithm uses only distance measurements between objects and between an object and a node. For example, if a node is represented as a minimum bounding rectangle (MBR), it computes the minimal distance between a point and that rectangle. The algorithm also works for generalized search trees (GiST) with arbitrary distance measures.
  - **Proof of correctness:** The priority queue orders entries by increasing distance. Because nodes use minimum bounding regions, every child node or object has a distance to the query that is equal to or greater than its parent node's distance. Thus, when a data object is at the top of the queue, every unvisited node and all their children and descendants are at least as far from the query and therefore not closer than that object at the top of the queue.
  - **Proof of optimality:** Assuming we already know the nearest neighbor of the query object $\boldsymbol{q}$, we define the Nearest Neighbor Sphere (NN-Sphere) as the circle centered at $\boldsymbol{q}$ that passes through that nearest neighbor. To be correct, the algorithm must consider all nodes that intersect the NN-Sphere, that is, nodes that contain points closer to $\boldsymbol{q}$ than the current nearest neighbor, because they might hold a better answer. In the example, the red rectangle must be considered, while the blue circle does not need to be checked.
  - The algorithm visits nodes in ascending order of their distance to $\boldsymbol{q}$. When an object reaches the top of the queue, it is the nearest neighbor. We can continue the search to find the k nearest neighbor by repeating the steps until k points were found at the top of the queue.



- Grid cells and minimum bounding regions approximate data point positions. These approximations let us set lower and upper bounds on distances and similarities for objects inside those regions. This makes it possible to apply the nearest neighbor algorithm to any structure that underestimates distances or overestimates similarity for data points within larger "containers". The figure below illustrates this idea for the L_2 metric and the dot product.



  - To compute lower and upper bounds on distances between a query and a minimum bounding region, the shape of the regions plays a crucial role. In the case of spheres and L_2-distances, we derive the lower and upper bounds by taking the distance between the query and the sphere's center and then subtracting (lower bound) or adding (upper bound) the sphere's radius (lower bound is zero if query is within sphere). Likewise, we can establish bounds on the dot-product of points within a sphere (as shown in the right-hand figure). These bounds are determined by the hyperplane that is orthogonal to the query vector and touches the sphere on both sides.
  - We often consider rectangular bounding regions and utilize $L_1/L_2/L_p$-distances and dot-products. Because these measures rely on component-wise operations, we can determine lower and upper bounds by minimizing and maximizing these component-wise operations. Let q_j represent the query component for dimension j, and l_j and u_j define the rectangle's boundaries. For the L_2-distance, we establish boundaries as follows (if the query component falls in between the rectangular boundaries, the lower bound contribution is 0):

$$lBnd(q, MBR) = \sqrt{\sum_{j=0}^{d-1} \begin{cases} (l_j - q_j)^2 & q_j < l_j \\ (q_j - u_j)^2 & q_j > u_j \\ 0 & l_j \leq q_j \leq u_j \end{cases}}$$

$$uBnd(q, MBR) = \sqrt{\sum_{j=0}^{d-1} \begin{cases} (u_j - q_j)^2 & q_j \leq \frac{l_j + u_j}{2} \\ (q_j - l_j)^2 & q_j > \frac{l_j + u_j}{2} \end{cases}}$$

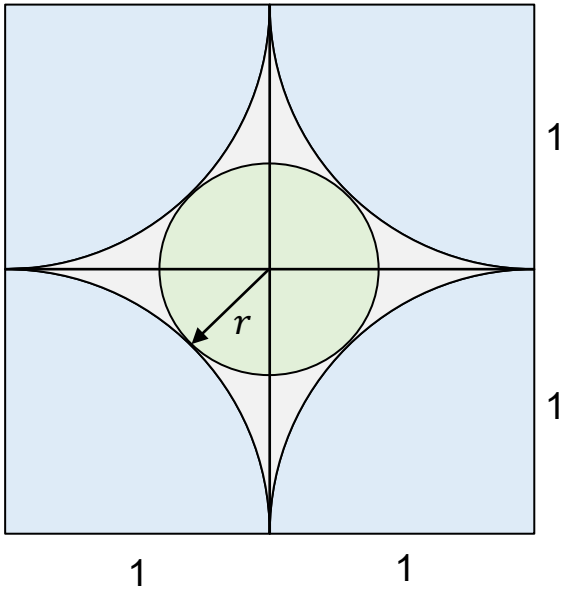
- Similarly, we can under- and overestimate dot-product similarities as follows:

$$lBnd(q, MBR) = \sum_{j=0}^{d-1} \begin{cases} q_j \cdot l_j & q_j \geq 0 \\ q_j \cdot u_j & \mathrm{q_j} < 0 \end{cases} \qquad uBnd(q, MBR) = \sum_{j=0}^{d-1} \begin{cases} q_j \cdot u_j & q_j \geq 0 \\ q_j \cdot l_j & \mathrm{q_j} < 0 \end{cases}$$

- In conclusion, Voronoi diagrams, gridfiles, and R-trees are all important spatial data structures, each with its own set of advantages and applications. However, their performance can be significantly impacted as the dimensionality of the data increases, a challenge commonly referred to as the "curse of dimensionality".
  - Voronoi Diagrams are powerful tools for spatial partitioning and proximity analysis. They work effectively in lower dimensions, helping with tasks like nearest neighbor searches and spatial analysis. However, as dimensionality increases, Voronoi diagrams become less practical due to the rapid expansion of the space and the exponential growth in the number of regions and computational complexity.
  - Gridfiles provide a structured way to index multidimensional data, offering a balance between storage efficiency and query performance. They are useful for lower-dimensional data where grid partitioning can efficiently reduce search space. In higher dimensions, Gridfiles face limitations as the grid becomes overly fine, leading to extensive storage requirements for the dictionary and inefficient searches as we explain next.
  - R-trees are hierarchical spatial data structures that excel in organizing and searching spatial data. They are versatile and can be used for various types of spatial data, from points to polygons. While R-trees are suitable for moderate dimensions, they also struggle with high-dimensional data due to issues like overlaps and complex splits, making them less efficient as dimensionality increases.
- Spatial data structures are valuable tools for managing and analyzing data in lower dimensions. However, as dimensionality grows, they face serious limitations that makes them impractical for many of the scenarios in semantic search and similarity search in general. This phenomenon, known as the "curse of dimensionality", highlights the need for specialized techniques and data structures to handle high-dimensional data effectively.

7.4 Curse of Dimensionality

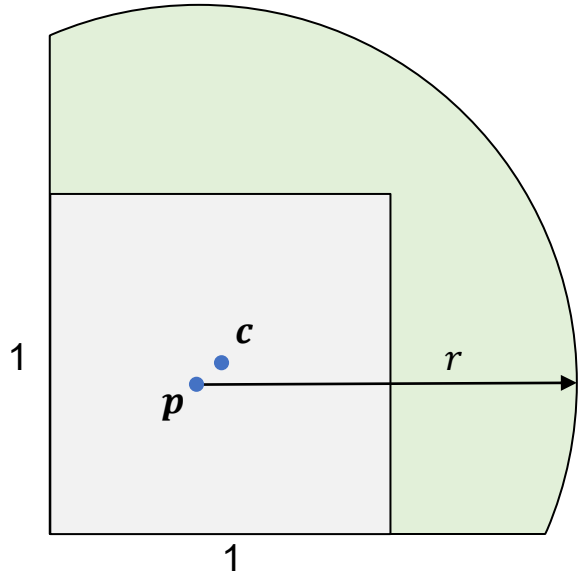
- The methods discussed earlier work well in lower-dimensional spaces, but as dimensionality increases, search performance declines noticeably. Beyond roughly 10 dimensions, a brute-force search, that is calculating distances or similarities between the query and every vector in the index, often outperforms more sophisticated methods, except in cases of heavy data clustering. This phenomenon is commonly referred to as the "curse of dimensionality", and many research efforts have attempted to overcome it, with limited success.
- Several studies show that performance declines as the number of dimensions increases. One major reason is that humans find high-dimensional spaces hard to understand, so search methods that work well in low dimensions often fail in higher ones. Properties that are easy to visualize and reason about in two or three dimensions, such as what is near and what is far, behave very differently in high-dimensional vector spaces. This changes how common measures work, such as Euclidean distance and dot-product similarity. Similar problems appear with other distance or similarity measures, but each case may need a slightly different explanation.
- High-dimensional spaces have several peculiarities that make search difficult. Distances between points tend to concentrate, meaning that the difference between the nearest and farthest neighbors becomes very small. This reduces the discriminative power of Euclidean distance. Dot-product similarities also tend to cluster around zero, making it harder to distinguish relevant matches. Other challenges include sparsity, where most points are far from each other, and increased storage and computational requirements, as each additional dimension adds complexity. Designing effective data structures and search algorithms in this context requires careful consideration of these high-dimensional effects.
- On the following pages we show a few oddities that appear when we apply our 2D and 3D geometric intuition to high dimensional spaces without care. Before you review the solutions on the right, pause and reflect on your own intuition and on how high dimensional geometry differs from everyday experience in lower dimensions.



The mysterious escape of the green sphere

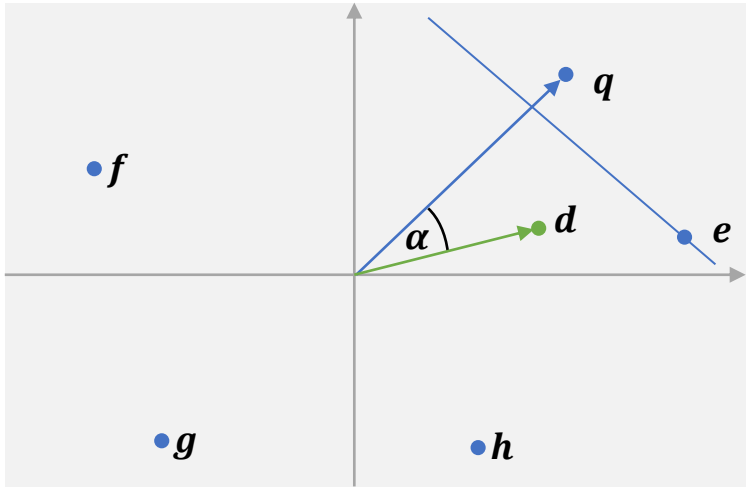
- Examine the illustration:
  - The large square has a side length of 2.
  - Each corner of the square has a circle with radius 1, covering all edges of the square.
  - In the center of the square, there is a circle with a radius r that touches all corner circles, captured inside the square
- What is the radius r of the inner circle?
- What happens when we apply the same scheme in a d-dimensional space, with hyperspheres at each corner of the hyper-cube, and a centered (green) hypersphere that touches all corner hyperspheres?

- The radius r of the inner circle is found by subtracting the corner circle's radius from the diagonal of one quadrant in the square: $r_2 = \sqrt{2} - 1 = 0.414$.
- We can extend this formula to a d-dimensional space. The diagonal of a d-dimensional hypercube with side length 1 is represented as $\sqrt{d}$. Given the corner spheres with radius 1, we can compute the radius of the central sphere as $r_d = \sqrt{d} - 1$.
- Let's now apply this to a 100-dimensional space: $r_{100} = 9$. In other words, the inner hypersphere, while still just touching all corner hyperspheres, has a significantly larger radius than what we previously considered the "outer" hypercube.
- In fact, the central hypersphere extends beyond the boundaries of the hypercube, even though we initially thought that the corner hyperspheres covered all the edges of the square/hypercube, preventing the inner hypersphere from reaching beyond.
- In a 3D setup, each of the 6 sides resembles the 2D version on the left (without the green sphere in the middle). As we can observe, there is a substantial space at the center of each side that remains uncovered. In higher dimensions, each of these $(d-1)$-dimensional facets becomes progressively less covered by the corner spheres, allowing the central hypersphere to extend beyond the hypercube while still just touching the corner hyperspheres.
- At $d = 4$, the green sphere is touching the hypercube, and with $d > 4$ it escapes the hypercube.



Big is never big enough

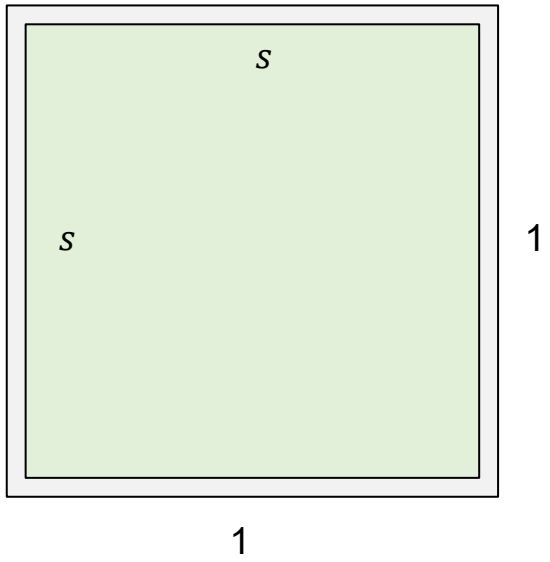
- Consider the illustration:
  - The gray square has a side length of 1.
  - $\boldsymbol{c}$ is at the center of the square with $\boldsymbol{c} = (0.5, 0.5)$.
  - $\boldsymbol{p}$ is slightly off-center within the square, located at $\boldsymbol{p} = (0.4, 0.4)$.
  - $\boldsymbol{p}$ serves as the center of a circle with a radius r (choose a large value for $r > 1$) that significantly extends beyond the square's boundaries.
- What is happening if we increase dimensionality and keep the radius r constant? Assume that $\boldsymbol{c}$ and $\boldsymbol{p}$ have in all dimensions the same component value.
- Let's choose a radius of $r = 3$ which is large enough to extend far beyond the square/hypercube
- The distance between $\boldsymbol{p}$ and $\boldsymbol{c}$ is expressed as $\delta = 0.1 \cdot \sqrt{d}$. As dimensionality increases, the distance δ increases proportionally to $\sqrt{d}$, and at a certain dimensionality d^*, δ surpasses the radius r.
  - $\delta = 0.1 \cdot \sqrt{d^*} > r$, that is if $d^* > 100 \cdot r^2$ the point c is farther away from p than the radius r
  - With $r = 3$, d^* is 900, which is a typical number of dimensions for features extracted from content.
- What does this imply? Despite our intuitive sense that $\boldsymbol{p}$ and $\boldsymbol{c}$ are close, the distances grow with $\sqrt{d}$ for the Euclidean distance. The maximum distance within the hypercubes (the diagonal) also grows at a rate of $\sqrt{d}$. However, the differences between the maximum and minimum distances do not increase as rapidly.
  - Let d_{min} and d_{max} represent the minimum and maximum distances between a fixed number of data points. "Fixed" means that the number of data points does not grow exponentially with dimensionality.
  - It can be proven that as the dimensionality d approaches infinity, the limit of $(d_{\max} - d_{\min}) \,/\, d_{\min}$ goes to 0. This makes Euclidean distances less suitable for high-dimensional spaces because all points seem to be equally distant from a query point.
- One approach to mitigate this is to use a different distance measure, such as the Manhattan distance, or work with fractional L_p metrics with $p < 1$.



What means "same direction"?

- Consider the illustration:
  - We have embeddings for the objects $\boldsymbol{d}$, $\boldsymbol{e}$, $\boldsymbol{f}$, $\boldsymbol{g}$, and $\boldsymbol{h}$. A query embedding $\boldsymbol{q}$ is provided.
  - For the cosine measure, data point $\boldsymbol{d}$ has the highest similarity due to its smallest angle to the query.
  - For the dot product, data point $\boldsymbol{e}$ holds the highest similarity as it lies on a hyperplane orthogonal to $\boldsymbol{q}$ and none of the other points are positioned "above" this plane.
- Generally, with a large number of data points, we can discover strong matches in the same direction (in this case, the upper right quadrant) of the query.
- How about in high-dimensional spaces?

- If we assume a "fixed" number of data points that doesn't grow exponentially with dimensionality, we observe the following:
  - Embeddings contain both positive and negative components, and the origin point divides each dimension into two parts.
  - In a 2D space, this results in 4 quadrants, each with unique sets of dimensions having positive and negative values. In a d-dimensional space, we have 2^d such areas.
  - We can interpret "same direction" as being in the same area as the query that is having the same sign for each component compared to the query.
  - In lower dimensions, with 1 billion data points, it is highly likely that many data points share the same sign for all component values.
  - However, in a 100-dimensional space, there are 2^{100} possible combinations of sign distribution across query components, making it exceedingly unlikely to find a data point with exactly the same signs.
- In higher dimensions, the notion of "same direction" weakens, as it's unlikely for any two points to have the same sign across all their components.
- It has been demonstrated that uniformly distributed points become orthogonal to each other as dimensionality increases. This renders both the dot product and cosine measures unsuitable for high-dimensional spaces, like embeddings with several hundred dimensions.

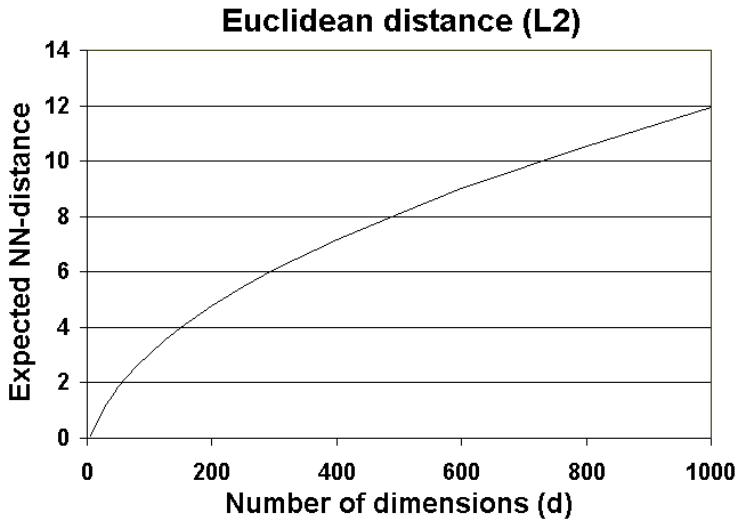


Where have all the data points gone?

- Consider the illustration:
  - We have two squares here. The outer square has a side length of 1, and the inner square has a side length of s, where $s < 1$. The inner square is centered within the outer square.
  - Let's choose $s = 0.99$. The inner, green square covers most of the outer square, leaving only a small portion near the outer square's edge uncovered.
  - When data points are evenly distributed in the data space, it's reasonable to assume that the majority of data points are inside the inner, green square, with only a few exceptional points located outside.
- Now, how does this change in higher dimensions?

- Let's consider a uniform distribution of data points in a d-dimensional space:
  - By design, in each dimension, there is a 99% probability that a randomly chosen data point falls within the boundaries of the inner square, leaving only a 1% chance that it resides outside these limits.
  - The probability of finding a data point within the inner square is determined by its volume, which is s^d. As dimensionality increases, the volume of the inner square decreases to zero with $s < 1$. With $d = 1000$, this results in a volume of $4.3 \cdot 10^{-5}$.
  - If we have 1 million data points, only 43 of them are inside the inner square; the majority are located outside. But where exactly are they?
- Even though data points are expected to reside within the inner square's range in 99% of each dimension, as the number of dimensions increases, there's a greater likelihood that, in at least one dimension, a data point falls outside the range: $P = 1 - 0.99^{1000} = 0.99996$.
  - This means it's highly probable that a data point is located near the edge of the outer square in at least one dimension. With $2d$ facets, most data points are within 0.005 units of distance from one of the facets.
  - In other words, we can conclude that in a high-dimensional space, the central region is largely empty, and data points are situated near the edges, within a small distance from one of the $2d$ facets.
- This also implies that any division of space by an index structure results in subspace volumes that rapidly approach zero probability of containing data.

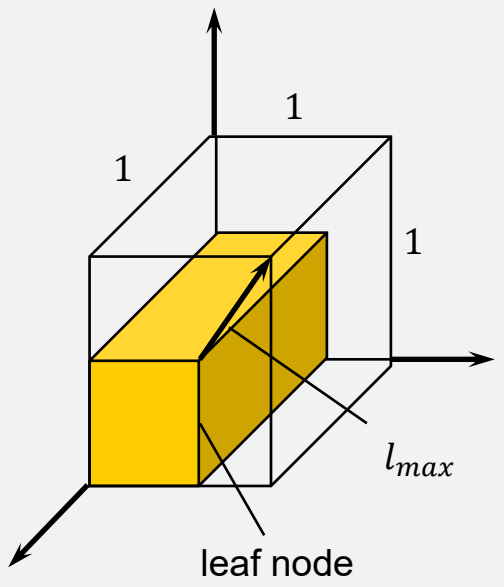
- Considering these observations, let's examine the behavior of index structures that partition space, such as the gridfile or the R-tree.
  - **Normalization:** To simplify the math, we assume a closed data space in the shape of a hypercube of side length 1 . We also assume independent dimensions and uniform distributions along these dimensions, which eliminates the need for dimensionality reduction. We only consider Euclidean distance for this analysis.
  - **Observation:** In the context of this space, the probability that a data point falls within a subspace is equivalent to the volume of that subspace. The total volume of the space remains 1, regardless of dimensionality.
  - **Gridfile:** As dimensionality grows, we encounter challenges with the cell-based approach. With increasing dimensions, we have at least 2^d cells, and even at moderate dimensionality, numerous cells remain empty. For example, with 1 billion data points and $d = 10$, most of the 2^{100} cells are empty. In addition, the dictionary size expands with 2^d, potentially exceeding available memory. A possible solution is to limit the number of dimensions involved in cell splits.
- In the following, we estimate the costs associated with nearest neighbor (NN) searches in hierarchical structures. To do this, we will first calculate the expected distance between the query point and its nearest neighbor. Then, we will make an estimation of the average number of leaf nodes/grid cells retrieved during the search.
  - Since we are utilizing the optimal NN-search algorithm, we can identify the leaf nodes to be accessed as all nodes that intersect with the NN-sphere around the query point.
  - Expected NN-Distance: The expected NN-distance represents the average distance between a query point and its nearest neighbor. The figure on the right side shows the expected NN-distance as it grows with dimensions (for a fixed data set).
  - With the expected NN-distance, we can define a sphere around a randomly selected query point with that distance as the radius. If this sphere intersects with the minimum region of a leaf node, we must visit that node during the NN-search.
  - Finally, we compute the probability that we need to visit, for a random query, a node in the index structure.



- **Number of leaf nodes / grid cells to visit:**
  - As discussed, it is not feasible to split all dimensions in the gridfile or similar structures utilizing rectangular MBRs. The core reason behind this limitation is that each additional split along a dimension results in a halving of volumes and, in turn, a decrease in the likelihood of a point residing within that MBR. For instance, when dealing with 1 million data points and $d = 14$, the expected number of data points drops to roughly 60 after splitting. While it is possible to perform further splits in dimensions, leaf nodes become increasingly less populated, and their numbers grow almost exponentially with each additional split until each leaf contains exactly one data point
  - Considering the number of leaf nodes to visit, we make the following assumptions: if the tree uses rectangular MBRs, it undergoes splits along d' axes, always dividing in the middle. Consequently, the MBRs of leaf nodes assume the shape shown in the figure on the right below. Let $l_{\max}$ represent the maximum distance between a point in the space and such a leaf node. Given the MBR's shape, this distance can be calculated as $l_{max} = 0.5 \cdot \sqrt{d'}$.
  - Now, let's compare this distance with the expected NN-distance, which leads to intriguing insights. When $d = 40$, $l_{\max}$ is approximately the expected NN-distance. Furthermore, for $d = 100$, $l_{\max}$ is much smaller than the expected NN-distance. This is due to the limited number of splits and ensuring we maintain non-empty leaves.

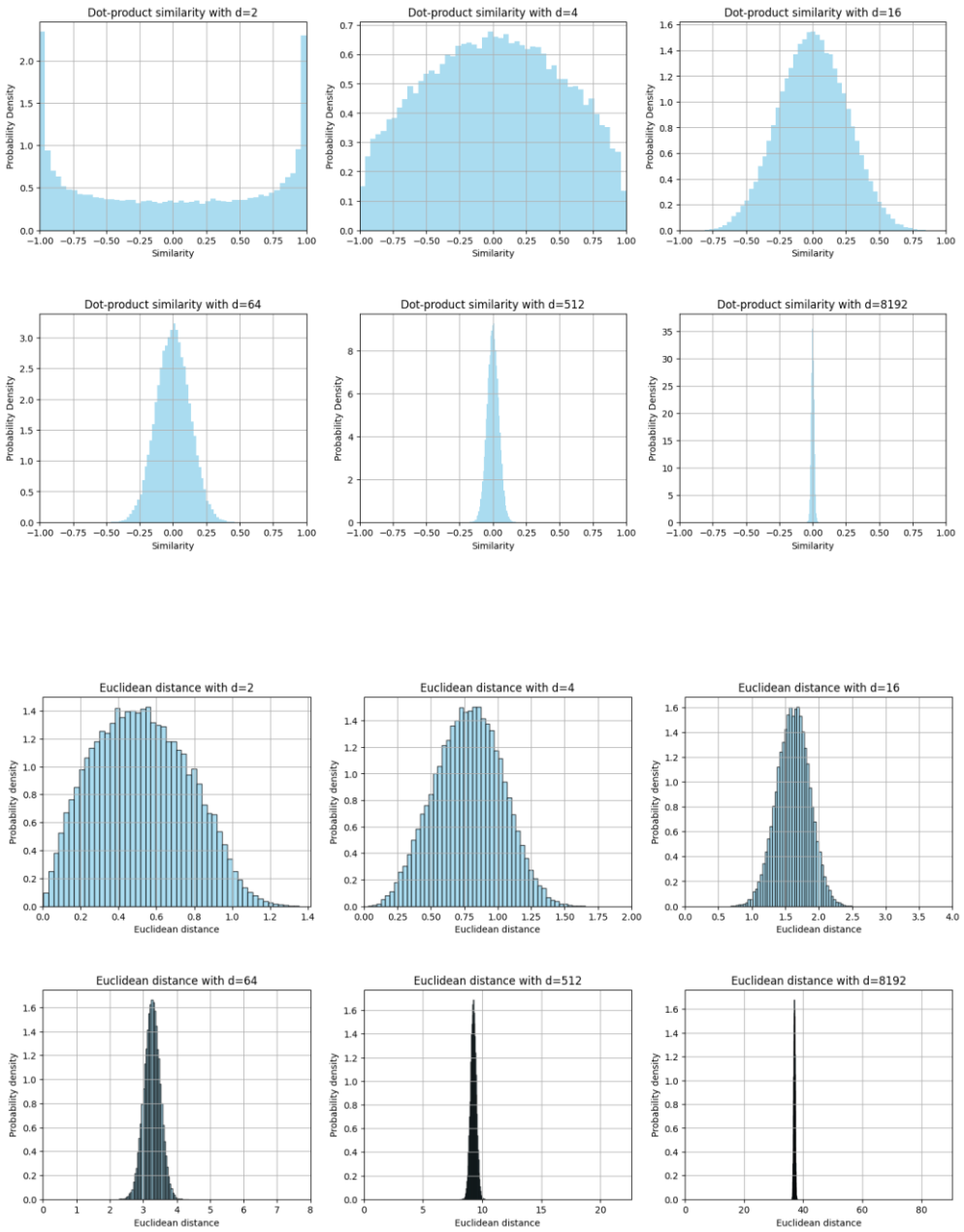
d	*N*	*d'*	l_{max}	NNdist
40	10^6	14	1.87	1.80
100	10^6	14	1.87	3.00

  - When $l_{\max} < NNdist$, a query point lies closer to all leaf nodes than to its nearest neighbor. Consequently, the MBR of each leaf intersects with the NN-sphere. Therefore, an optimal NN-search must visit all leaves to obtain the nearest neighbor for any query in the data space.



- This situation raises an important question: why do we employ hierarchical structures if we ultimately need to access all data?
  - It can be shown for most hierarchical data structures, that beyond a relatively small number of dimensions (say 20-50), the method no longer performs better than a brute force search for the nearest neighbor.

- Finally, let us examine how dot product similarities and Euclidean distances change as dimensionality increases. Both sets of distance and similarity distributions were obtained through a Monte Carlo simulation using pairs of random vectors. For the dot product, component values lie within the range [-1, 1], and all vectors are normalized so that they lie on the unit hypersphere. For the Euclidean distance, component values lie within [0, 1], and vectors are not normalized.
  - With the dot product and dimensionality $d = 2$, similarity scores range from -1 to 1, and many pairs of points have high similarity values above 0.95. As dimensionality increases, the distribution changes sharply. At $d = 8192$, nearly all similarity values cluster around 0. As noted earlier, in very high dimensions most points become almost orthogonal, meaning their dot products approach 0. The small differences between values make it increasingly difficult to distinguish between good and poor matches.
  - A similar pattern appears with the Euclidean distance measure. In low dimensions, such as $d = 2$, distances span a wide range, making it easy to separate nearby and distant points. As dimensionality increases, for example at $d = 8192$, most distances concentrate around a mean value of 36.9 with a standard deviation of 0.24. This means that 99 percent of the distances fall between 36.22 and 37.67, making distance an unreliable indicator of how well an object matches a query.

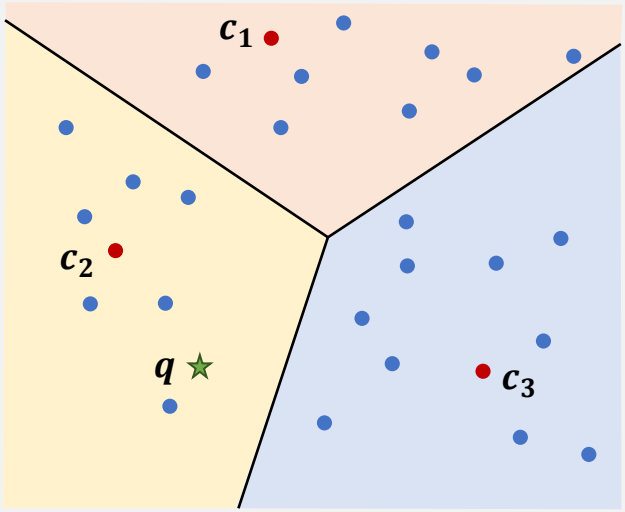


7.5 Approximate NN-Search

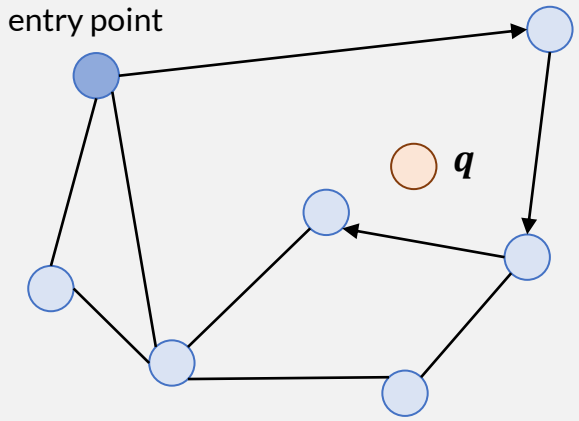
- For small datasets, doing a full vector scan, which means comparing the query embedding to every stored embedding, works well and gives exact results. As the dataset grows to millions or billions of vectors, this exhaustive method becomes computationally infeasible. Comparing every vector soon exceeds practical limits for both time and memory.
  - To address this scalability challenge, researchers began exploring approximate nearest neighbor (ANN) methods in the 1990s. One early approach was Locality Sensitive Hashing (LSH), which replaced exact search with an approximate one. The main idea of LSH and later approaches is to **trade a small amount of accuracy for a large gain in speed**, letting systems find results that are good enough much faster than an exact search.
  - In many modern applications, such as embedding search for retrieval-augmented generation (RAG) systems, finding the exact nearest neighbors is often unnecessary. What matters is retrieving a few relevant documents or passages that together have enough information to answer a user's query. Approximate methods usually achieve this and are far more efficient.
  - Early work on speeding up search focused on reducing vector dimensionality, for example using Principal Component Analysis (PCA) or Singular Value Decomposition (SVD). More recent embedding designs, like Matryoshka embeddings, get similar efficiency by arranging information hierarchically inside the vector space. Component-wise quantization encodes each vector component with a small number of bits. This causes a small approximation error, but in practice it does not change relative distances between vectors enough to affect nearest neighbor rankings.
- Even with data reduction methods, we still must search through all the data, although we read less than for an exact search. In this chapter, we provide an overview of modern nearest neighbor and similarity search methods, using Facebook AI Similarity Search (FAISS) as an example. FAISS brings together the main techniques developed for nearest neighbor (NN) and most similar (MS) search, so it applies across many domains. It uses a composite index structure that combines several complementary methods into one efficient search index. FAISS implementations are also highly optimized for both CPU and GPU architectures.
  - FAISS includes the traditional methods described above and adds more recent improvements. For example, the SQ4 index performs approximate search by quantizing each component to 4 bits, while the Flat index does a brute force scan of all vectors. Applying PCA beforehand can reduce dimensionality. You can combine these methods with HNSW or IVF to greatly reduce the number of items that must be read.

- The FAISS framework organizes the search process into four main phases:
  - **Vector Transformers:** Techniques for normalizing and transforming vectors. The choice of a vector transformer depends on the data, the distance or similarity measure you want, and the indexing method you will use. Some transformations prepare vectors for indexing, for example by normalizing for cosine measures, while others provide a rough initial result by reducing dimensionality.
    - **L2-norm**: Normalizes vectors by their L2 norm, which is beneficial for both cosine similarity and maintaining consistent distance and similarity values.
    - **PCA**: Applies principal component analysis to reduce dimensionality, typically to around 64 dimensions. This focuses vectors on the most discriminating directions and preserves the utility of distances/similarities.
    - **Optimized Product Quantization (OPQ)**: Optimizes vectors for subsequent product quantization, reducing quantization errors and improving estimates.
    - **Padding**: Adjusts the dimensionality of vectors to align with the requirements of a subsequent method.
  - **Non-exhaustive search components (coarse quantizers):** Methods that use coarse-grained selection to reduce the amount of data considered during search, allowing early elimination of candidates.
    - See the following pages for implementations in the framework, such as HNSW and IVF.
  - **Encodings (fine quantizers):** Quantization techniques that lower the computational burden in estimating distances or similarities to the query.
    - See the following pages for implementations in the framework, such as LSH and PQ.
  - **Refiners:** While vector transformers prepare data before searching, refiners process results afterward. In many approximate search cases, refiners are unnecessary, and the outcomes of the approximate methods are used directly to generate user search results. However, there are situations where refiners can be beneficial:
    - Refiners re-rank search results using extra criteria. One method uses the original vector data to compute exact distances or similarities between the query and each item. To improve accuracy, we retrieve the vector representations of the top $n = m \cdot k$ results and pick the best k matches. A parameter lets us balance higher search cost against better result quality.
    - Some methods do not compute scores directly, but we need those values to combine scores from different methods. In those cases, calculating distances or similarities for the top results gives the search engine and the user the required information. For example, we add extra constraints or predicates on metadata and apply them during retrieval. As the index returns more results, we filter out items that do not meet these criteria.

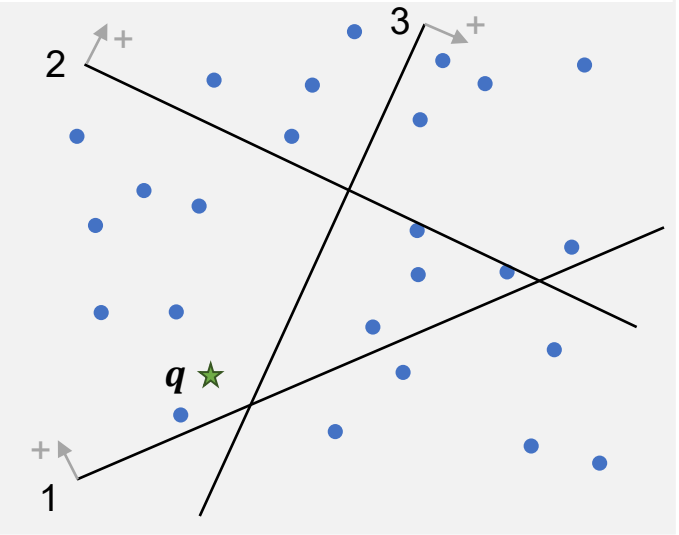
- **Non-exhaustive search components (coarse quantizers):** The **Inverted File (IVF)** reduces search scope through clustering of data. In principle, we could use any scheme to cluster data, but we discuss here a k-means based clustering. This method is simple, fast and provides good results.
  - Consider the figure below: we can choose a number of clusters (e.g., $\mathrm{n} = 4096$), and the k-means algorithm selects n centers, $\boldsymbol{c_1}, \ldots, \boldsymbol{c_n}$ that minimize the sum of distances between data points and their closest center.
  - Each data point is assigned to the nearest center, and the n centers serve as a coarse-grained representation for all the data points assigned to them (indicated by the amber, red, and blue areas in the figure). These n centers effectively cluster the dataset, providing a quantization code for each data point in the collection.
  - The inverted file is essentially a dictionary with the n centers, each of which maps to a list containing the data points assigned to the respective clusters of these centers.
  - With a query $\boldsymbol{q}$, we determine the closest center among the n centers (for example, in the figure, $\boldsymbol{c_2}$ is the closest center). While it is probable that the nearest neighbor to $\boldsymbol{q}$ is in the same cluster, it is not guaranteed. Still, we reduce the search to the cluster of the nearest center to obtain an approximate result, even though it may miss points in nearby other clusters. By utilizing an inverted list, we scan through the list linked to that cluster, effectively reducing search time by a factor of $1/n$ (e.g., $1/4096$) at the cost of reduced quality (assuming a good balance of the k-means clustering with about equal data points per cluster).
  - To enhance search accuracy, we can explore $m < n$ clusters based on the distances between the query and their centers. This involves scanning m lists, increasing search time to m/n while improving result quality.
  - Each list in the inverted file can be further indexed using additional methods to speed up the search within those lists. For example, a product quantizer can be used to estimate approximate distances. However, stacking multiple approximate methods can result in a rapid decline in result quality. Thus, it is crucial to choose hyperparameters thoughtfully to strike the right balance between time savings and quality preservation.



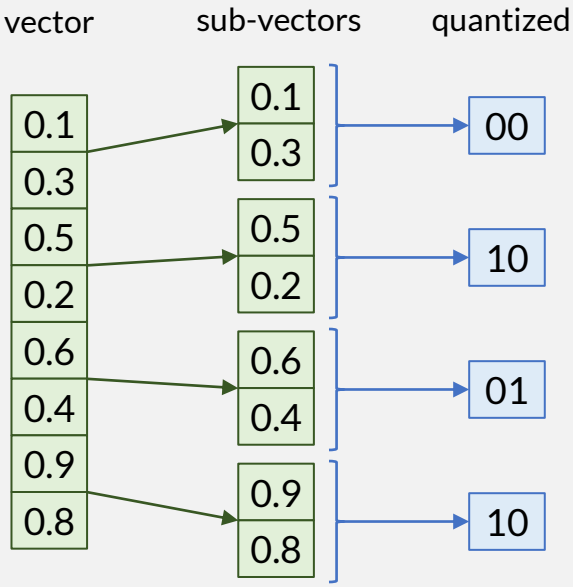
- **Non-exhaustive search components (coarse quantizers):** The **Hierarchical Navigable Small World (HNSW)** can work stand-alone or in combination with a product quantization method to enhance navigation speed.
  - A Navigable Small World (NSW) is a graph structure with two key properties:
    - **Navigability**: NSW allows for efficient and effective traversal from one data point to another within the high-dimensional space. This navigability is achieved by creating connections between data points in a way that reflects the underlying geometric structure of the data, making it easier to find nearby neighbors during search.
    - **Small-World Property:** In high-dimensional spaces, most data points can be reached through relatively short paths (poly-logarithmic). This property is crucial for efficient nearest neighbor searches, enabling quick location of relevant data points without full dataset traversal.
  - In the lower-right figure, the graph consists of data points as nodes, with each node having a fixed number of connections or so-called "friends." Friends are chosen from the nearest neighbors within the dataset. Additionally, shortcuts to other areas are introduced to enhance navigation and prevent isolated sections. While the number of connections per node can differ, NSWs generally work towards maintaining balanced connections.
  - During a search, we start at specific entry points in the graph. For each node, we use a greedy navigation strategy by following connections that lead us closer to the query point. If we arrive at a node where all connected nodes are farther from the query, it indicates a local minimum in the graph, and we can consider this as an approximate answer. The search process for finding the most similar data point (dot-product) operates in a similar manner.
  - The HNSW hierarchy consists of multiple layers: the base layer encompasses all data points, forming a full NSW structure. Above the base layer, multiple higher layers are established, each with its own NSW, using well-distributed data points. Each higher layer uses fewer data points and contributes to a search hierarchy: at higher layers, coarse regions of interest are identified. Navigating down to the base layer, we refine nearest neighbors with new connections from the lower layer.
  - HNSW can be combined with other techniques to reduce the cost of distance calculations, such as product quantization or PCA. It is one of the fastest methods for approximate search and is included in many frameworks to provide approximate search. Different implementations use different strategies to build and tune the graphs.



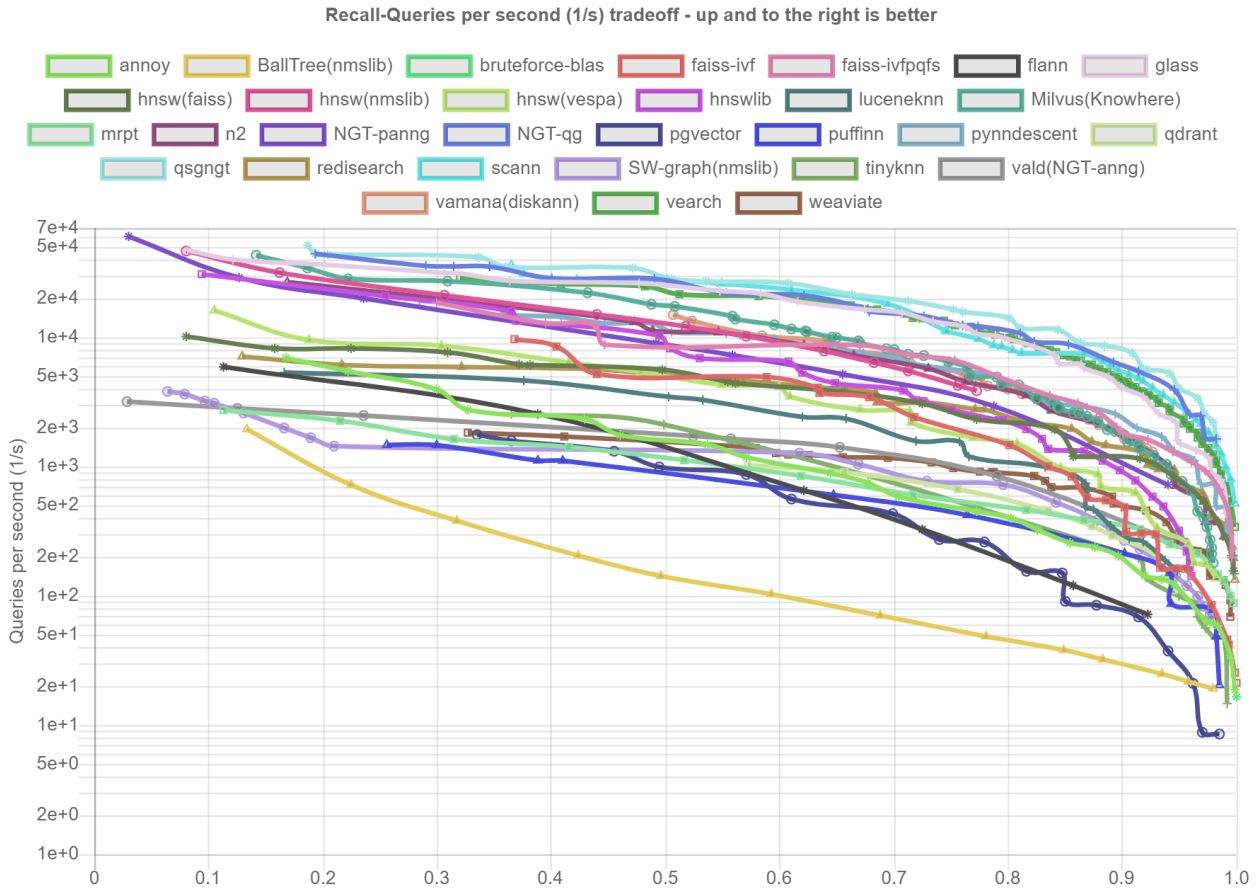
- **Encodings (fine quantizers):** The **Locality Sensitive Hashing (LSH)** uses n hyperplanes to represent the position of a data point with regard to each hyperplane. LSH encodes the position with a n-bit string: if a data point is located on the positive side of hyperplane j, it is assigned a value of 1 in the j-th position of the bit-string, and 0 otherwise. Consequently, the n hyperplanes encode data points' spatial positions with 2^n potential combinations.
  - We can choose the hyperplanes randomly or employ a method to select an optimized set of hyperplanes. In the lower-right figure, we defined three hyperplanes that partition the data space into seven areas (one of the eight potential combinations is not possible). Each area is represented by a 3-bit string, where a "1" indicates that the area is above the corresponding hyperplane. For instance, the position of the query $\boldsymbol{q}$ can be described as "100" because it only lies above hyperplane 1.
  - As an alternative, we can use the principal axes of the data space to define d hyperplanes. The bit string assigned to data points is then determined by the signs of their component values (1 for positive values, 0 for negative values).
  - We then use these bit-strings to quantize the data points that share the same representation. However, with even a moderate number of hyperplanes, the number of partitions exceeds the number of data points, resulting in many partitions being empty and others being sparsely populated.
  - Instead, we calculate the Hamming distance between the query's bit-representation and the bit-representations of all data points. Then, we choose the k data points with the smallest Hamming distance as an approximate search result. Since we rely solely on the Hamming distance, this approach is most effective when not too many data points share the same representation.
  - If many points have identical Hamming distances, an additional refinement steps becomes necessary to re-rank data points. During reranking, we retrieve the actual vector representations, calculate precise similarity/distance values, and arrange data points accordingly. We can combine this method with an inverted files method and apply the LSH on each list separately to further accelerate the search.
  - LSH offers an effective balance between search quality and retrieval costs. By adjusting the number of bits, we can control the retrieval speed and memory usage as a hyperparameter. For data points in a d-dimensional space and with the choice of n hyperplanes, we can reduce the data size from $4 \cdot d$ bytes to $n/8$ bytes, resulting in an equivalent speedup for the calculation of Hamming distances.



- **Encodings (fine quantizers): Product Quantization (PQ)** is an extension of the component-wise quantization method. In PQ, vectors are divided into m sub-vectors, each with a dimensionality of d/m. These sub-vectors are usually non-overlapping, and often require that the total dimensionality, d, is a multiple of m. Otherwise, we can employ padding to extend vectors to the next multiple of m.
  - In the figure at the bottom, an 8-dimensional vector is divided into $m = 4$ sub-vectors. Each sub-vector is quantized into a 2-bit string. When concatenated, this compresses the 8 floating-point numbers into 8 bits, achieving a 1:32 compression ratio in this example.
  - For each sub-vector, we employ a separate quantization scheme. Instead of scalar quantization, we typically use k-means clustering (or other quantization methods). This divides the sub-space into 2^n clusters, where n represents the number of bits for each sub-vector. The cluster centers approximate the spatial locations of the data points.
  - At query time, we first compute the distances between the query's sub-vectors and all cluster centers. In our example, this requires $m \cdot 2^n$ sub-vector distance calculations. Subsequently, we scan through the data points, utilizing the n-bit strings for each sub-vector as lookup values for the previously computed distances to the centers. Essentially, for each sub-vector, we employ the cluster center as an approximation for the query-to-data distance in that sub-space. Summing up all distances across sub-vectors provides the overall similarity/distance, and we use these values to identify the k best answers.
  - While scalar quantization optimizes each dimension individually, product quantization optimizes across multiple dimensions. Moreover, it allows for greater data compression and the assignment of fewer bits than there are dimensions within that sub-vector. Achieving this level of compression is not possible with scalar quantization, unless dimensions are skipped.
  - The speedup of PQ is due to the use of small values for n, typically 8, 12, or 16, which determine the size of the codebooks representing the distances to cluster centers. Larger values for n create excessively large codebooks that may not perform well on certain CPU/GPU architectures. In such cases, increasing the value of m can be necessary.
  - PQ can be combined with techniques like inverted files (IVF), where IVF initially narrows the search scope, and each of its lists is encoded with a separate PQ index. PQ then expedites the scanning of these lists to obtain the final answer.



- The website https://ann-benchmarks.com offers a comparison of the performance of approximate search methods on real-world datasets. It also offers a benchmarking framework for testing your own method and submitting results for publication. The figure below displays the results for a 100-dimensional GloVe embeddings dataset with $k = 10$.
  - The x-axis represents recall, which, in this context, signifies the proportion of approximate top-10 answers that are also present in the top-10 of an exact search.
  - The y-axis displays the query throughput, indicating the number of queries per second achievable at a specific recall threshold.



7.6 Frameworks for Vector Search

- **Lucene** provides native nearest neighbor search via the `KnnVectorField` and `KnnVectorQuery` classes, added in version 9. This lets you store and search fixed-length floating point vectors and supports both approximate and exact similarity search. The approximate search uses the HNSW (Hierarchical Navigable Small World) algorithm, which arranges vectors in layered small-world graphs to find nearest neighbors in sublinear time. Lucene can use cosine, dot product, or Euclidean (L2) similarity depending on how the field is configured. Each Lucene segment keeps its own HNSW graph and query results are merged across segments.
  1) **Create index:** Creating the index works the same as before. You can set the analyzer, the directory, and other settings.
  2) **Inserting data:** When adding data, set each field type as usual. You can use `KnnVectorField` to store vector data in the index. Lucene's configuration automatically builds the HNSW structure as vectors are added.
  3) **Perform Query**: A `KnnVectorQuery` object defines the vector search given a query embedding a the number of object to retrieve (k=5). It is combined with a range query (year < 2000) inside a `BooleanQuery`, mixing semantic similarity with structured filtering. The `IndexSearcher` executes both efficiently in a single pass.

```
// 1. define analyzer, directory, and index writer
Directory directory = FSDirectory.open(Paths.get("./index"));
Analyzer analyzer = new EnglishAnalyzer();
IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter writer = new IndexWriter(directory, config);


// 2. build a document from key-value data
Document createDocument(Map<String, String> data) {
   Document doc = new Document();
   doc.add(new TextField("title", data.get("title"), Store.YES));
   doc.add(new IntField("year", Integer.parseInt(data.get("year")),
Store.YES));
   doc.add(new FloatField("rating",
Float.parseFloat(data.get("rating")), Store.YES));
   doc.add(new TextField("body", data.get("body"), Store.NO));
  doc.add(new KnnVectorField("embedding", data.get("embedding")));

   return doc;
}

// 3. vector search with predicate
DirectoryReader reader = DirectoryReader.open(directory);
IndexSearcher searcher = new IndexSearcher(reader);

float[] query = new float[]{...};
Query vectorQuery = new KnnVectorQuery("embedding", query, 5);
Query yearFilter = IntPoint.newRangeQuery("year",
                                    Integer.MIN_VALUE, 1999);

BooleanQuery combinedQuery = new BooleanQuery.Builder()
                .add(vectorQuery, BooleanClause.Occur.MUST)
                .add(yearFilter, BooleanClause.Occur.FILTER)
                .build();
TopDocs topDocs = searcher.search(combinedQuery, 5);

System.out.println("Top results (year < 2000):");
for (ScoreDoc sd : topDocs.scoreDocs) {
    Document d = searcher.doc(sd.doc);
    System.out.printf("- %s (%s) | Score: %.4f%n",
                      d.get("title"), d.get("year"), sd.score);
}
```

- **OpenSearch** provides nearest neighbor (NN) search through the k-NN plugin. The plugin extends Lucene with data structures for high-dimensional vector similarity. OpenSearch mainly uses HNSW for approximate NN and brute-force for exact similarity. Supported similarity metrics include cosine, dot product, and L2 distance. A limitation is that vector search is separate from keyword or BM25 scoring. In a distributed cluster, each shard keeps its own HNSW index and results are merged at query time, which can slightly affect global ranking precision but scales well.
  1) **Creating the index**: The mapping lists fields and their data types and includes a `dense_vector` field with a set dimension (`dims`). Setting `"knn": true` enables the HNSW engine for the index.
  2) **Inserting data**: Data are bulk indexed into the cluster. Each record includes metadata (title, year, rating) and an embedding vector. Bulk insertion is more efficient because it batches requests and lets OpenSearch build the internal graph incrementally.
  3) **Performing the query**: The `bool` query applies a filter (year < 2000) and a `knn` clause that finds the embeddings most similar to the query vector. The `k` parameter controls how many neighbors are returned. The result combines structured filtering with semantic similarity and returns the top scoring items for output generation.

```
1) Create a new OpenSearch index

from opensearchpy import OpenSearch
from opensearchpy.helpers import bulk

# Connect to OpenSearch; adjust for your instance
client = OpenSearch(
    hosts=[{'host': 'localhost', 'port': 9200}],
    http_auth=('admin', 'admin'),
    use_ssl=False,
)

INDEX_NAME = "movies"


# Define index mapping with dense_vector
index_body = {
    "settings": {
        "index": {
            "knn": True  # Enable KNN search
        }
    },
    "mappings": {
        "properties": {
            "title": {"type": "text"},
            "id": {"type": "keyword"},
            "year": {"type": "integer"},
            "rating": {"type": "float"},
            "embedding": {
                "type": "dense_vector",
                "dims": 1024,  # e.g., Qwen3-0.6B
                "index": True,
                "similarity": "cosine"
            }
        }
    }
}

# Create new index
client.indices.create(index=INDEX_NAME, body=index_body)
print(f"Created index '{INDEX_NAME}'")
```

2) Insert movie collection

```
# Example data (you can replace this with your own)
collection = [
    {
        "id": "2",
        "title": "Inception",
        "year": 2010,
        "rating": 8.8,
        "embedding": [0.2, 0.1, 0.4, ...]
    },
    ...
]


# Bulk insert
actions = [
    {"_index": INDEX_NAME, "_id": movie["id"], "_source": movie}
    for movie in collection
]
bulk(client, actions)

print(f"Inserted {len(collection)} movies")
```

3) Perform vector + year constraint query

```
query_embedding = [0.15, 0.25, 0.45, ...]  # your query embedding

query_body = {
    "size": 5,
    "query": {
        "bool": {
            "filter": [
                {"range": {"year": {"lt": 2000}}}
            ],
            "must": [
                {
                    "knn": {
                        "embedding": {
                            "vector": query_embedding,
                            "k": 5
                        }
                    }
                }
            ]
        }
    }
}

response = client.search(index=INDEX_NAME, body=query_body)



# Print results
print("\nTop results (year < 2000):")
for hit in response["hits"]["hits"]:
    source = hit["_source"]
    score = hit["_score"]
    print(f"- {source['title']} ({source['year']}) | Score:
  {score:.4f}")
```

- **PostgreSQL** supports vector similarity search via the **pgvector** extension. It adds a vector data type for fixed-length arrays of floats. Vectors can be indexed with several approximate nearest neighbor methods: IVF, HNSW, or a brute-force scan. Exact search is still available for small datasets or high-precision needs. These indexes integrate with PostgreSQL's planner, so vector operators (for example `<->` for Euclidean and `<#>` for dot product) work alongside normal SQL filtering and sorting.
- Internally, PostgreSQL is a relational database, not a distributed system like OpenSearch. Vector indexes are stored per table, and queries run on a single node.
  1) **Create the index:** Before using vector search, enable the pgvector extension. The schema defines a `VECTOR(3)` column with a fixed dimension. In real cases, replace 3 with the actual embedding dimensionality. Then create an index on the vector data, specifying the index type and the distance measure. Note that pgvector works with distances, not similarities.
  2) **Insert data**: vector data is added as strings and then automatically indexed (here: the HNSW is expanded with the new vector).
  3) **Perform query**: A SQL query filters rows with `year < 2000` and orders results by cosine distance using the `<=>` operator. This combines semantic similarity with structured filtering directly in SQL.

```sql
-- 1) Enable pgvector extension and create table
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE movies (
    id SERIAL PRIMARY KEY,
    title TEXT,
    year INT,
    rating FLOAT,
    embedding VECTOR(3)  -- embedding dimension = 3
);


-- Create an approximate nearest neighbor index
CREATE INDEX ON movies USING hnsw (embedding vector_cosine_ops)
  WITH (m = 16, ef_construction = 64);



-- 2) Insert movie data
INSERT INTO movies (title, year, rating, embedding) VALUES
('The Matrix', 1999, 8.7, '[0.1, 0.3, 0.5]');



-- 3) Perform a vector similarity query with a filter (year <
  2000)
-- '<->' is the distance operator (smaller = better)
-- '<=>' is the cosine distance (smaller = better)
-- '<#>' is the negative dot product operator (smaller = better)

SELECT
    title,
    year,
    rating,
    embedding <=> '[0.1, 0.25, 0.45]' AS distance
FROM movies
WHERE year < 2000
ORDER BY embedding <=> '[0.1, 0.25, 0.45]'
LIMIT 5;
```

7.7 Literature and Links

Is NN-search meaningful in high-dimensional spaces

- 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), 1998. https://www.vldb.org/conf/1998/p194.pdf
- 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, 2001. https://bib.dbvis.de/uploadedFiles/155.pdf

Index structures for NN-search

- 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, 1984. https://dl.acm.org/doi/10.1145/348.318586
- A.Guttman. R-Trees: **A dynamic index structure for spatial searching.** In Proceedings of the ACM SIGMOD International Conference on Management of Data, 1984. https://dl.acm.org/doi/10.1145/971697.602266
- 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), 1996. http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-70774
- 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), 1997. https://www.vldb.org/conf/1997/P426.PDF

Approximate NN-search

- Y. Malkov, D. Yashunin, **Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs** (2016), IEEE Transactions on Pattern Analysis and Machine Intelligence. https://arxiv.org/abs/1603.09320
- Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. **Locality-sensitive hashing scheme based on p-stable distributions**. In Proceedings of the twentieth annual symposium on Computational geometry.2004 https://www.cs.princeton.edu/courses/archive/spring05/cos598E/bib/p253-datar.pdf
- H. Jégou, M. Douze, C. Schmid. **Product Quantization for Nearest Neighbor Search.** IEEE Transactions on Pattern Analysis and Machine Intelligence, 2011. https://inria.hal.science/inria-00514462v2/document

Implementations and Benchmarks

- **FAISS: Facebook AI Similarity search** https://github.com/facebookresearch/faiss
- **ANN Benchmarks**: https://ann-benchmarks.com and https://github.com/erikbern/ann-benchmarks/
- **pgvector**: https://github.com/pgvector/pgvector
- **OpenSearch k-NN query**: https://docs.opensearch.org/latest/query-dsl/specialized/k-nn/index
- **Lucene**: https://lucene.apache.org