CS 6400 A

Database Systems Concepts and Design

Lecture 13

10/08/25

Announcements

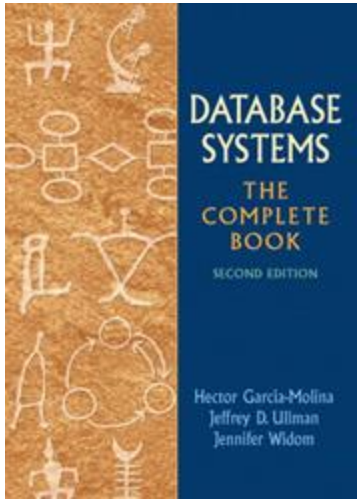
Assignment 2 released

- Due Oct 20

Project

- Never too early to start working on your project!
- Feedback on project proposal: expect next week
- Project Milestone: first week of November

Reading Materials



Database Systems: The Complete Book (2nd edition)

- Chapter 14.6: Tree Structures for Multidimensional Data

Reference papers

- HNSW
- Product Quantization

Agenda

1. Multi-dimensional Indexes and ANNS

2. Graph-based Methods

3. Product Quantization

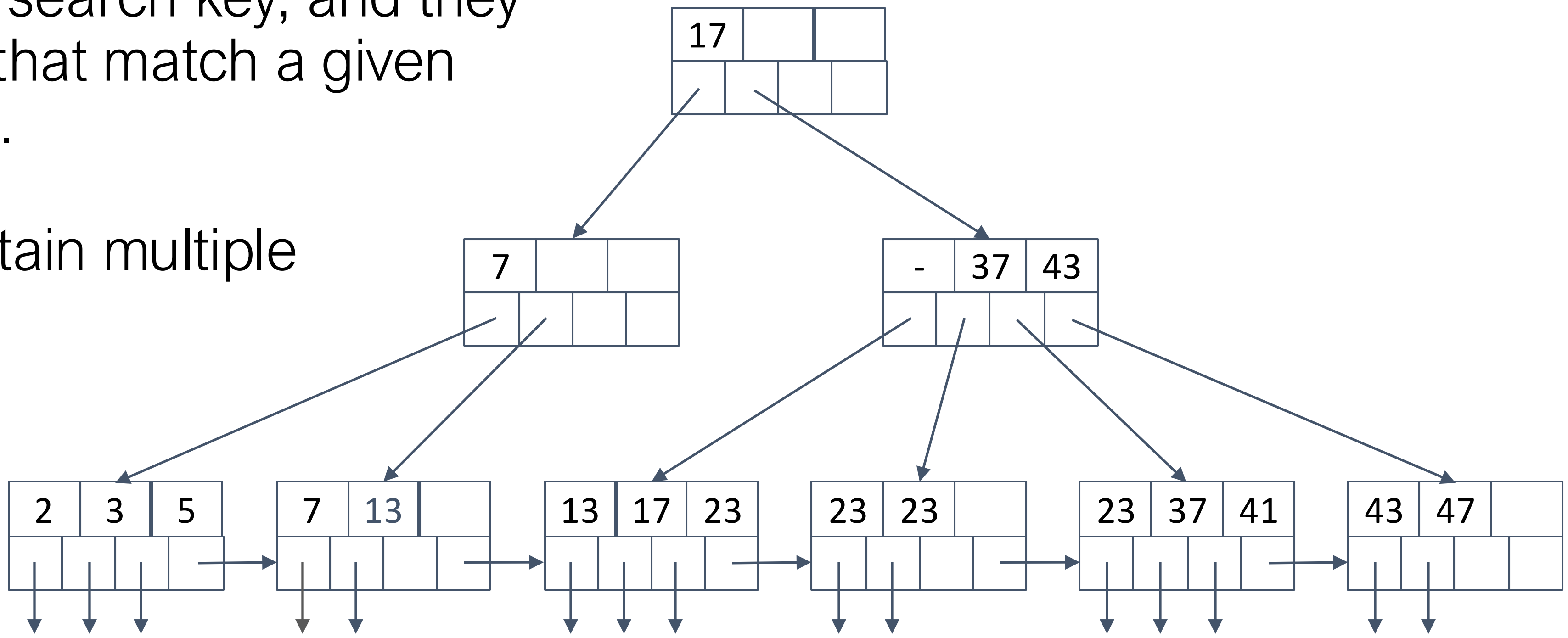
1. Multi-dimensional Indexes and ANNS

One-dimensional Indexes

Recall that **B-trees** are examples of a one-dimensional index

- Assume a single search key, and they retrieve records that match a given search key value.

- The key can contain multiple attributes



Limitation of 1D indexes

Example spatial queries:

- Find the 10 closest restaurants to my current location
- Find all coffee shops within a 1 km radius of my current location

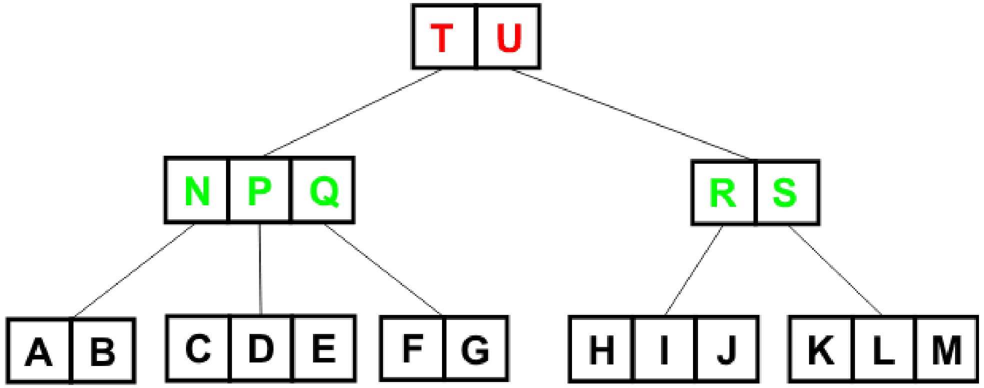
Building a B-tree on either the latitude or longitude is inefficient, since the query for a geographic area is essentially a range query in both dimensions *simultaneously*.

Multidimensional Indexes (Tree-based)

Multidimensional indexes:

- Examples: kd-tree, R-tree
- Specifically designed to partition multi-dimensional data

R-tree



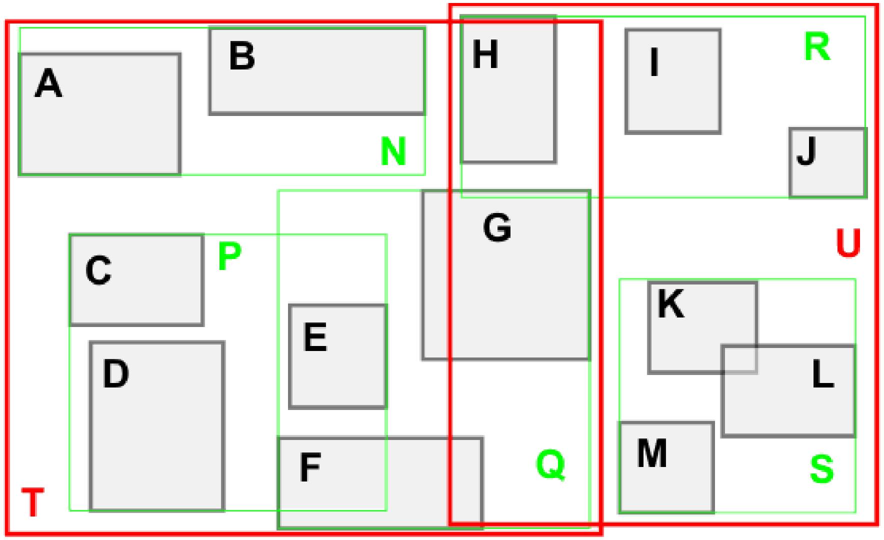


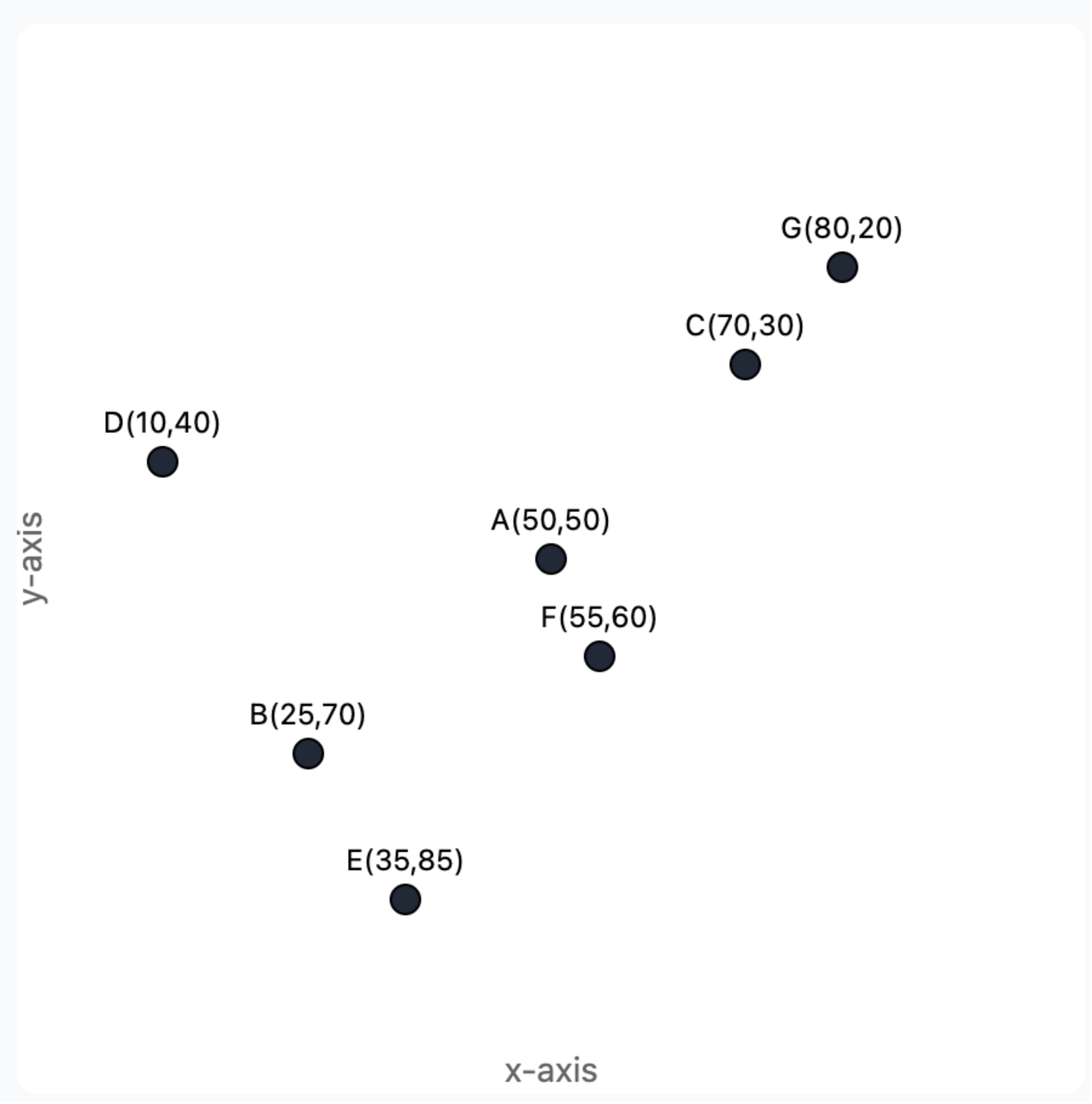
Image source: https://cglab.ca/~cdillaba/comp5409_project/R_Trees.html

KD-Tree

A KD-Tree is a binary search tree that cycles through dimensions; designed primarily for in-memory operations.

Construction Algorithm (High-Level):

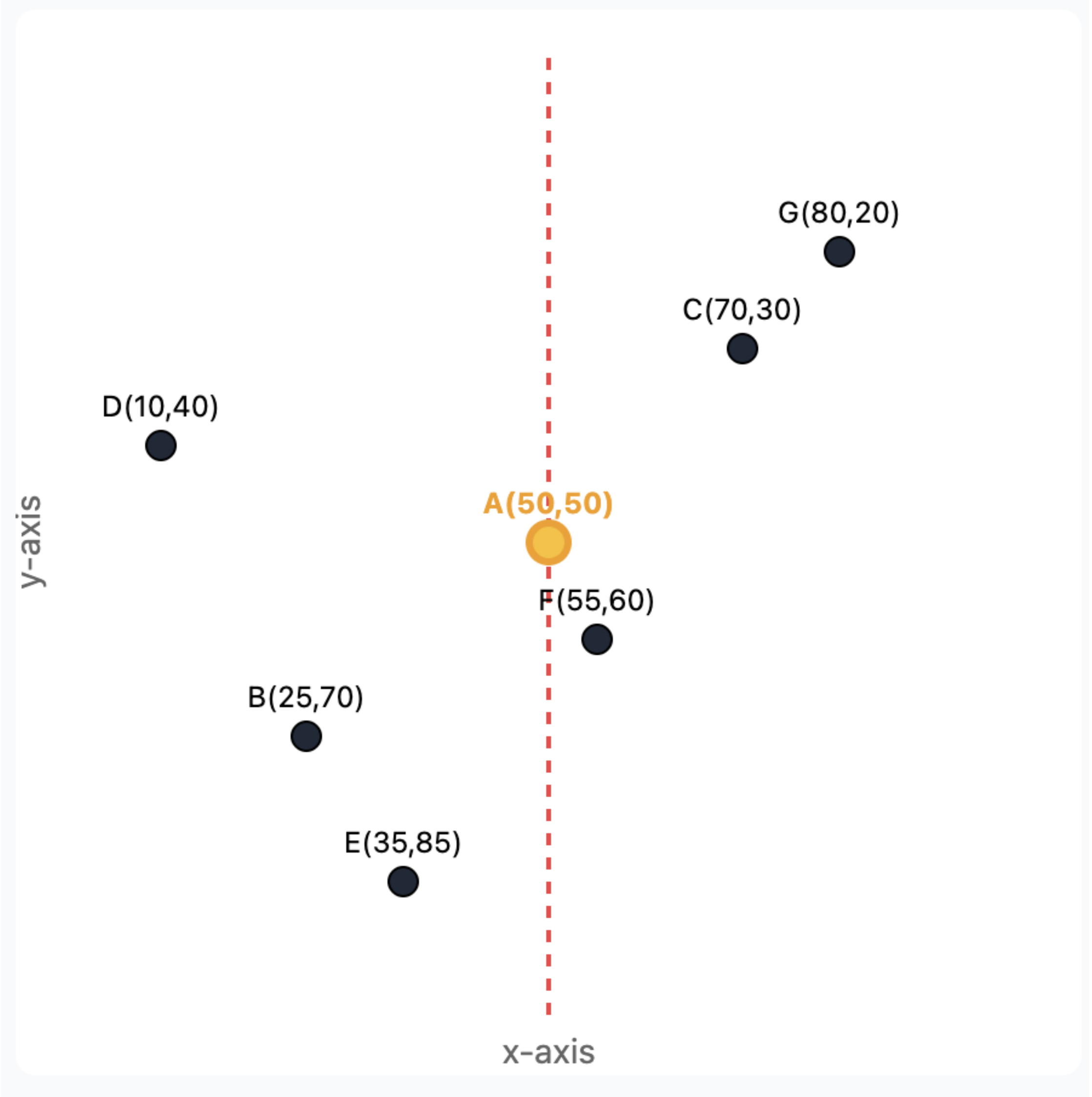
- Start with a set of points and choose a dimension to split on (e.g., round-robin).
- Find the median point in that dimension.
- Split the data into two subsets: {points ≤ median} and {points > median}.
- Recursively build the left and right subtrees, switching to the next dimension.



KD-Tree

A KD-Tree is a binary search tree that cycles through dimensions.

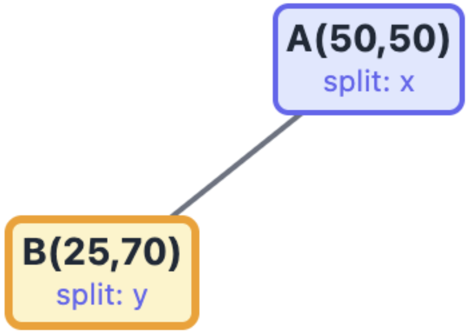
Tree Structure

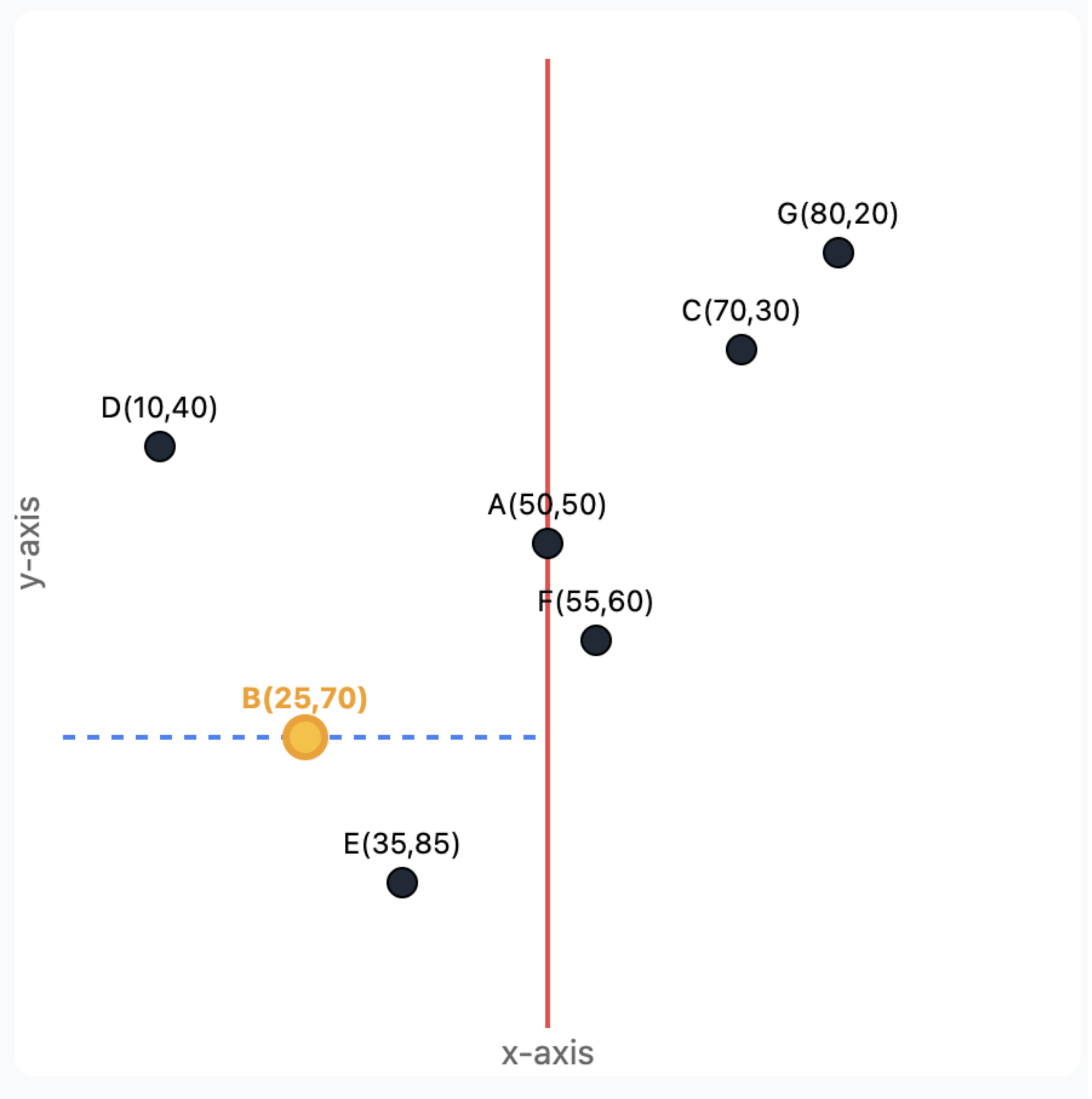


KD-Tree

A KD-Tree is a binary search tree that cycles through dimensions.

Tree Structure

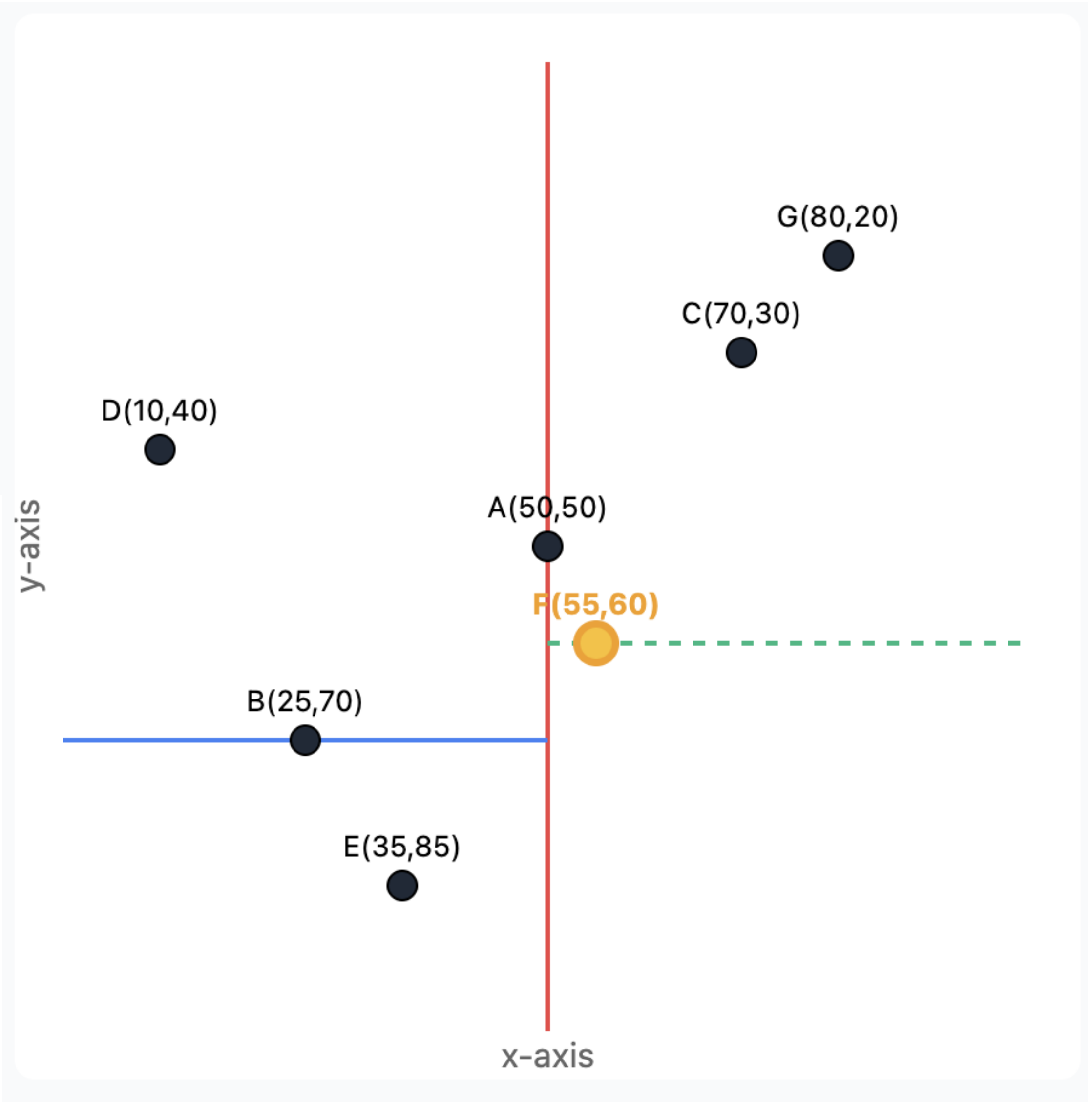




KD-Tree

A KD-Tree is a binary search tree that cycles through dimensions.

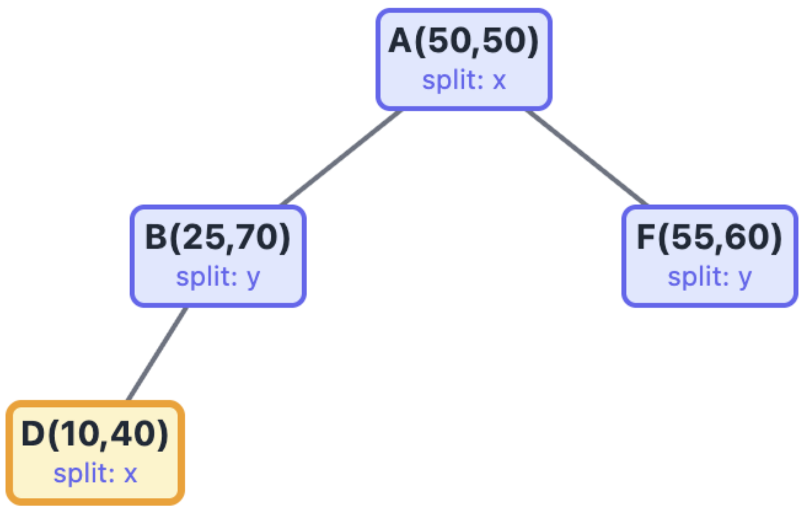
Tree Structure

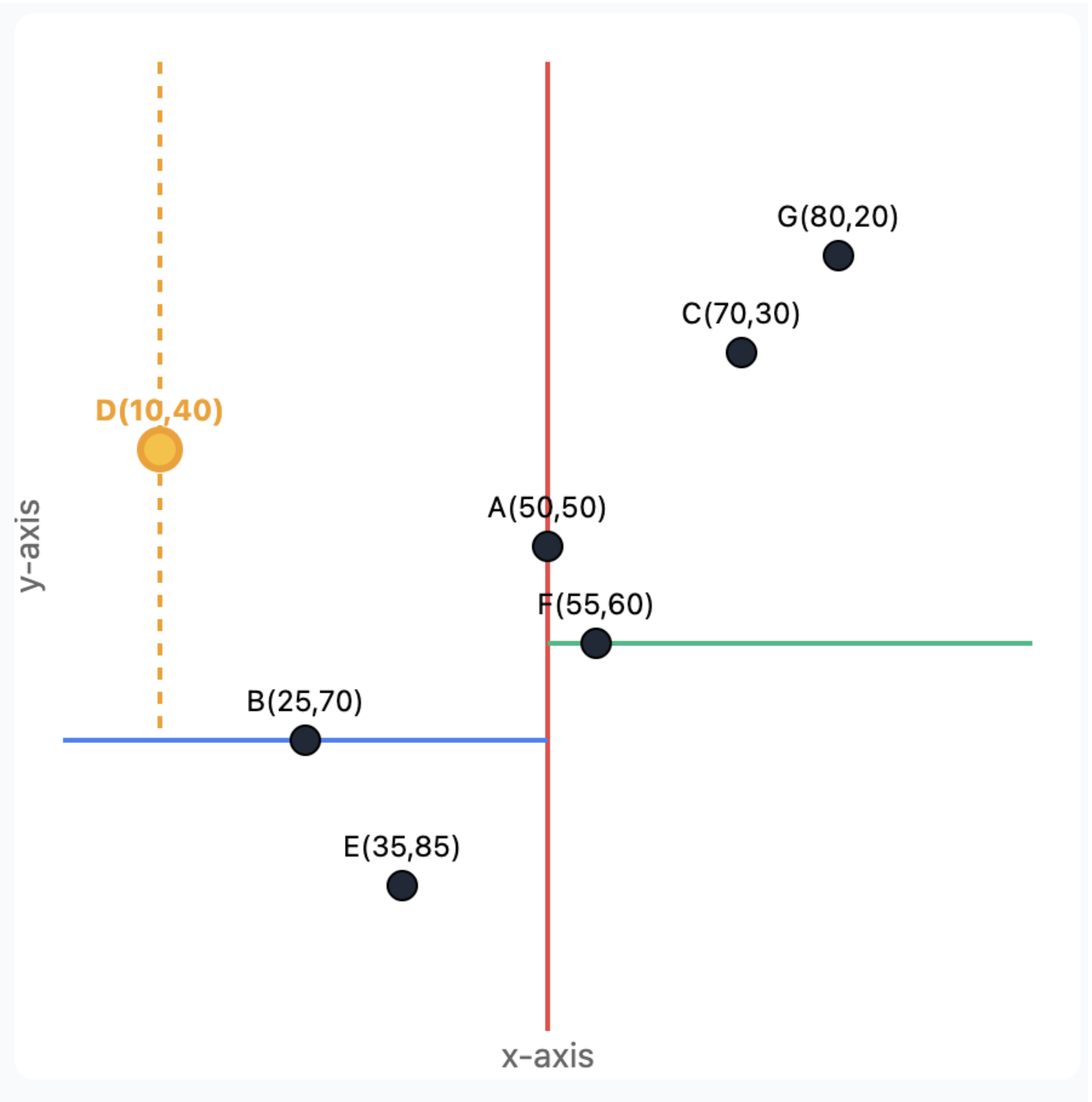


KD-Tree

A KD-Tree is a binary search tree that cycles through dimensions.

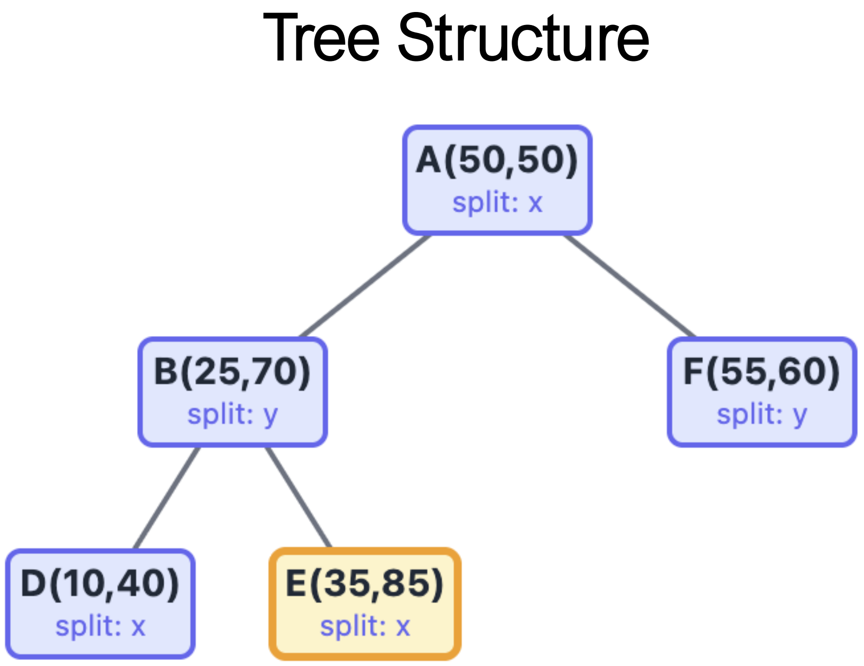
Tree Structure

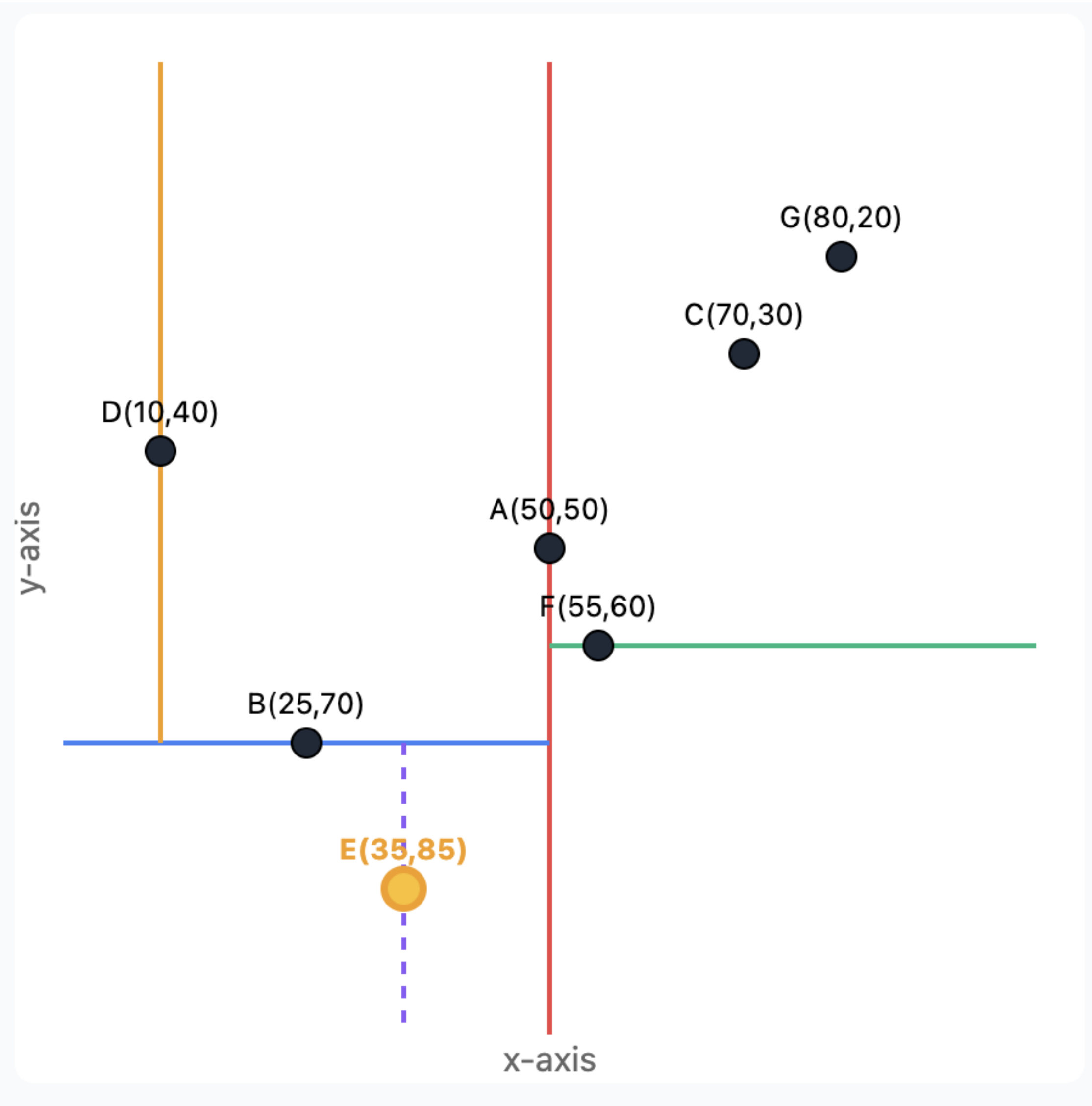




KD-Tree

A KD-Tree is a binary search tree that cycles through dimensions.





Nearest Neighbor Queries (kNN Query)

Given a query object q, we search in a high-dimensional dataset $\mathcal{D}$ for one or more objects in $\mathcal{D}$ that are among the closet to q according to some distance metric.

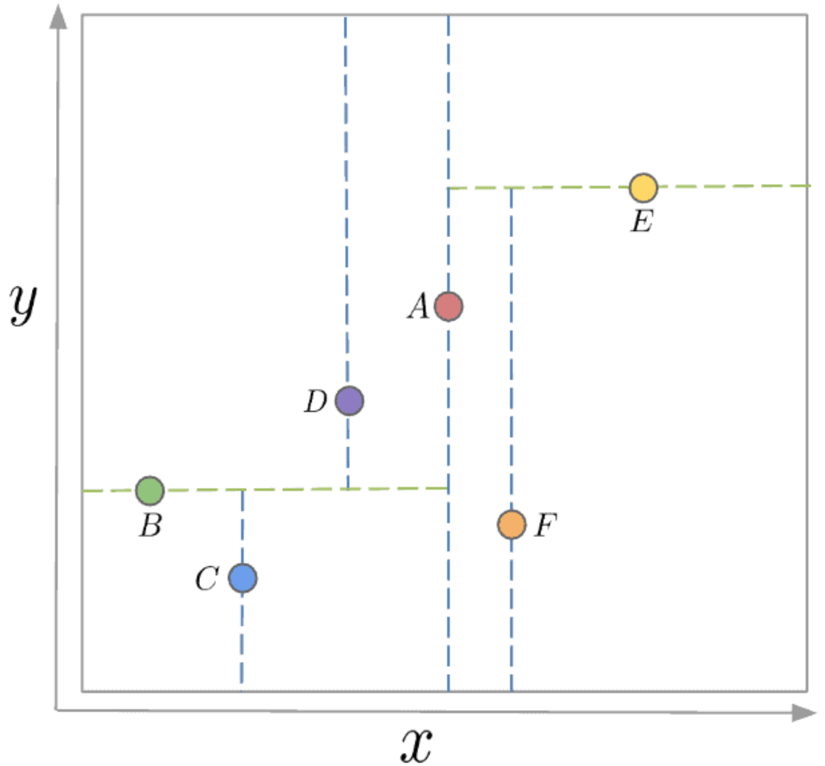
Common distance metric:

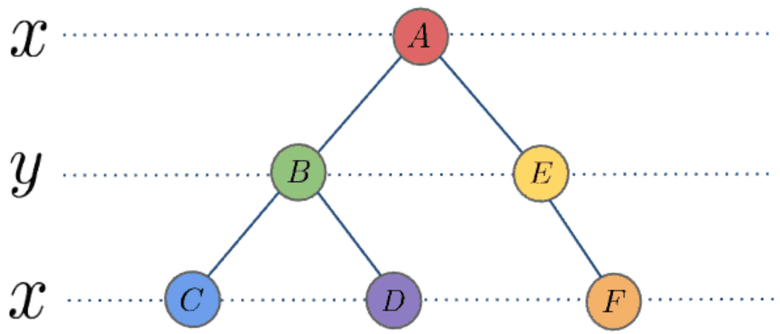
- Euclidean distance (supported by kd-tree): $||\vec{\mathrm{q}} - \vec{\mathrm{p}}||_2$
- Cosine similarity: $\frac{q \cdot p}{||q||\ ||p||}$
- Jaccard similarity: $\frac{|\mathrm{q} \cap \mathrm{p}|}{|\mathrm{q} \cup \mathrm{p}|}$ (q and p are two arbitrary sets)

KD-Tree

Search Algorithm (kNN Query):

- **Traversal:** Start at the root and traverse down the tree (comparing the query point to the split value at each node) until you reach a leaf. This is your initial "best guess."
- **Backtracking:** As you unwind the recursion, check if a *better candidate* could exist on the *other side* of the splitting hyperplane.
  - If the distance from the query point to the hyperplane is less than the current best distance, you *must* search the other subtree.



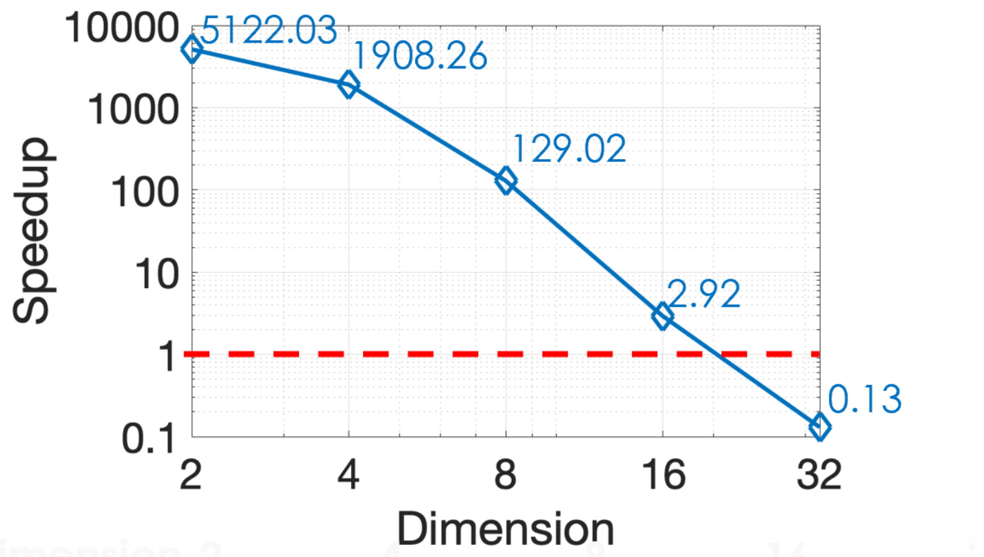


Curse of Dimensionality

Linear scan takes $O(n)$ per query; kd-tree takes $\sim O(n^{1-1/d})$

When the dimension d is very large, search trees (e.g., kd-tree, R-Tree) performs no better than the linear scan, due to the "curse of dimensionality" [C1994].

Example: k-d tree versus linear scan.



[C1994] K. L. Clarkson. An algorithm for approximate closest-point queries. In Proceedings of the Annual Symposium on Computational Geometry, pages 160–164, 1994.

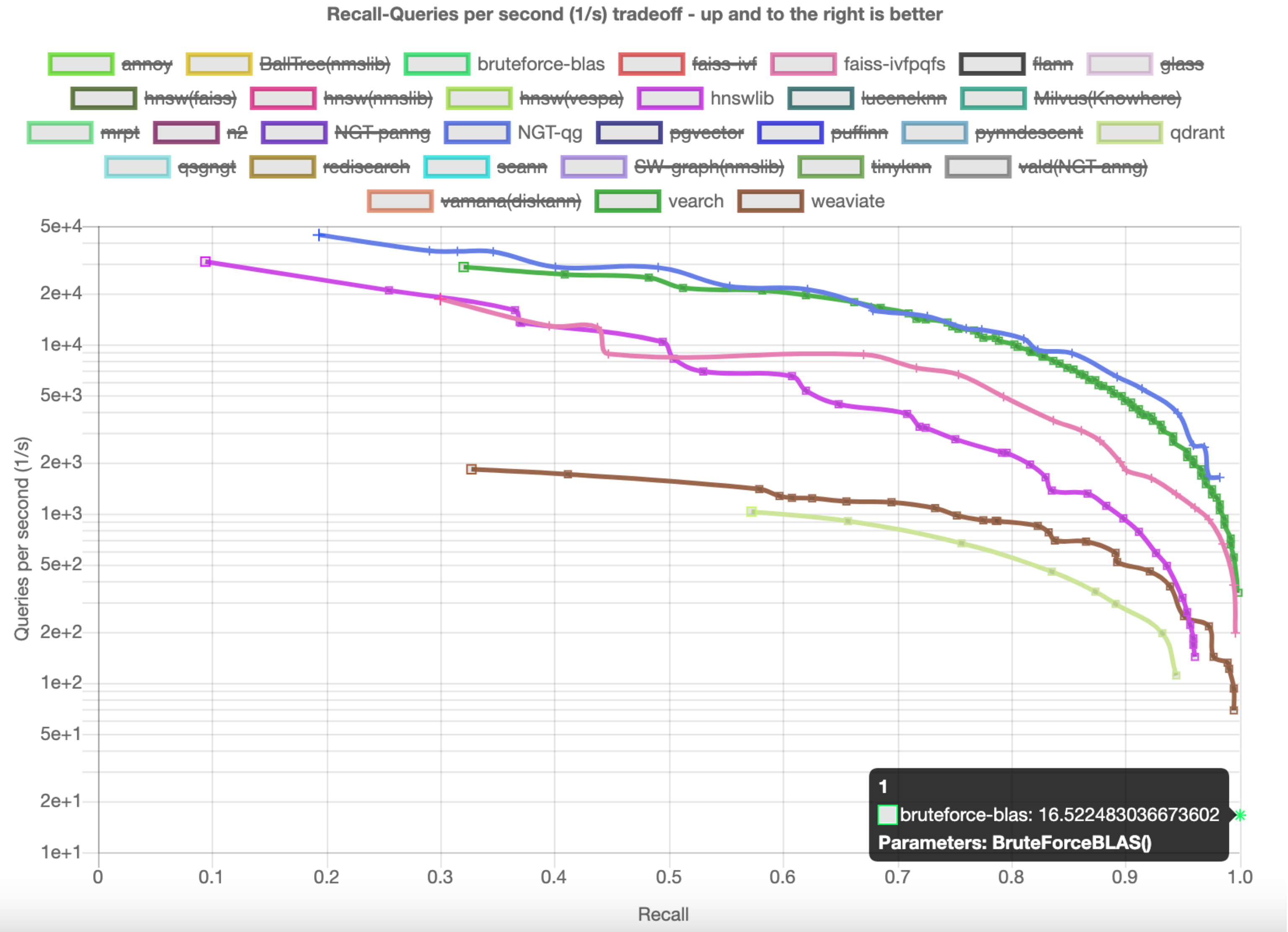
Approximate Nearest Neighbor Search

Problem Definition: Given a query object q, we search in a massive high-dimensional dataset $\mathcal{D}$ for one or more objects in $\mathcal{D}$ that are among the closet to q *with high probability* according to some similarity or distance metric.

ANNS solutions are usually much faster than linear scan with negligible accuracy loss.

- Tradeoff between performance and accuracy

Approximate Nearest Neighbor Search

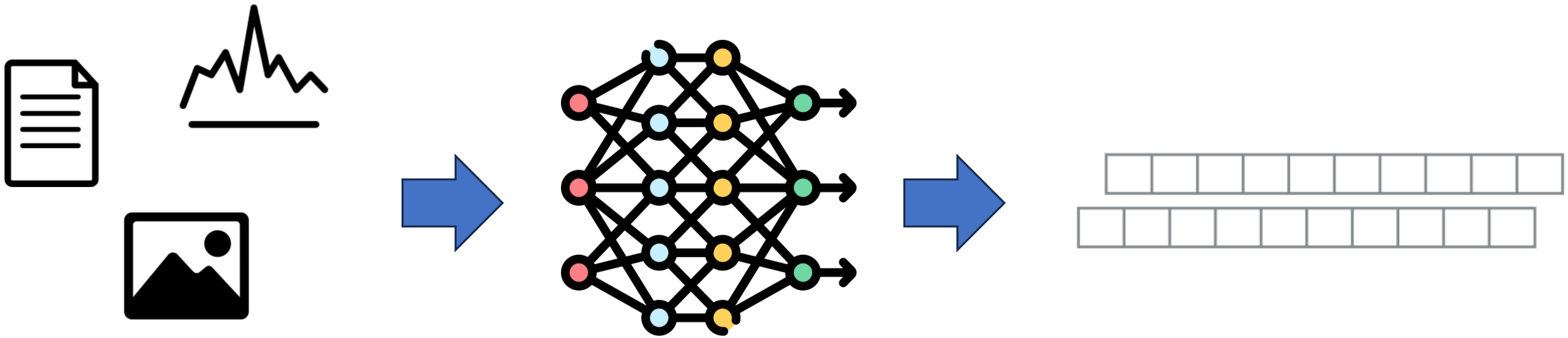


KNN search

This image is downloaded from: http://ann-benchmarks.com/fashion-mnist-784-euclidean_10_euclidean.html

Applications of ANNS

- Finding the most relevant data points in the database when compared to a specific query point

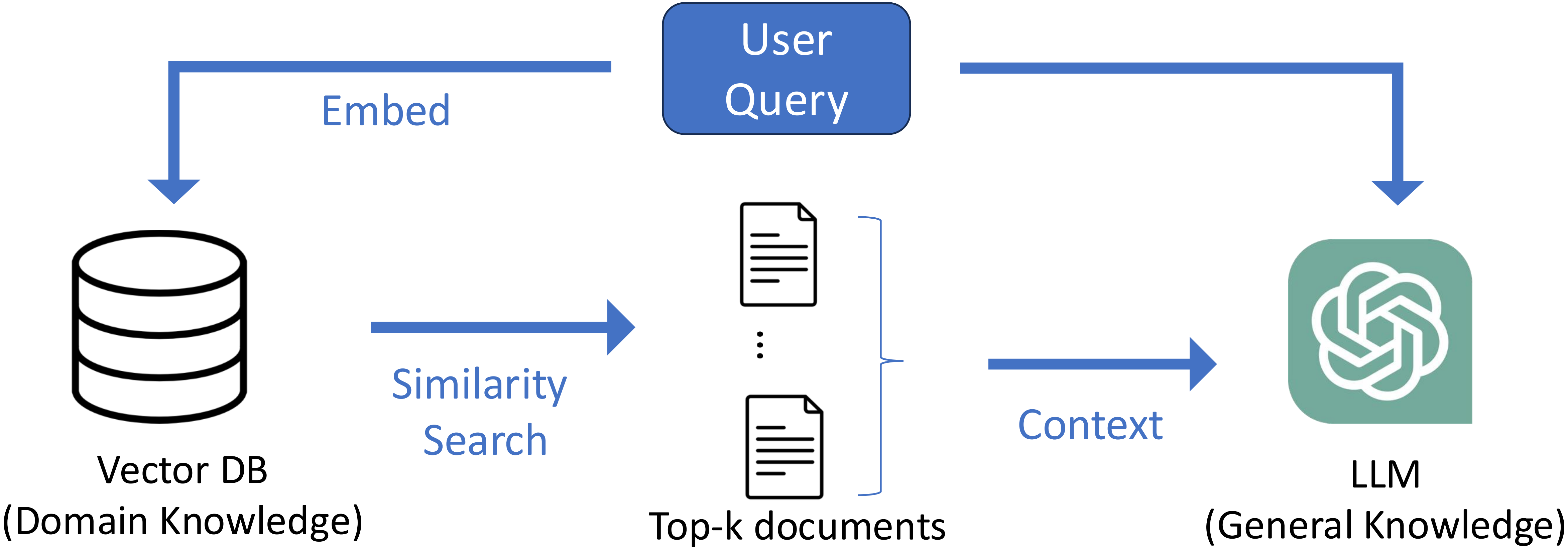


Structured and unstructured data

ML

embeddings

Example: Retrieval Augmented Generation



Scale of Embeddings

Example: OpenAI

- text-embedding-3-small: 1536 dims
  - 1536 * 4 bytes = 6 KB
  - 6 KB * 1B = 6 TB
  - 6 KB * 1T = 6 PB

- text-similarity-davinci-001: 12288 dims
  - 12288 * 4 bytes = 49 KB
  - 49 KB * 1B = 49 TB
  - 49 KB * 1T = 49 PB

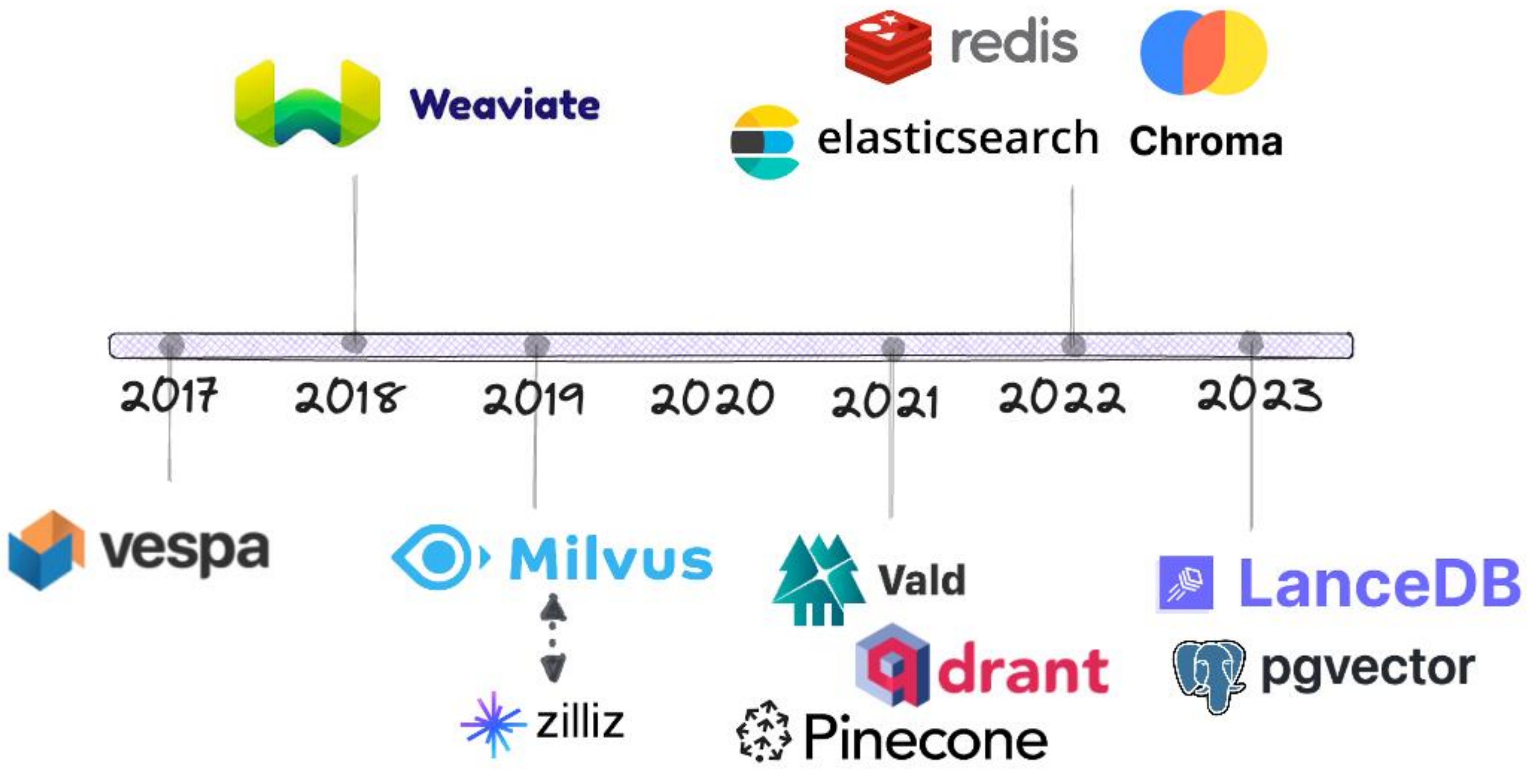
Significant memory requirement for processing billion/trillion scale vector datasets!

Source: openai.com

Vector Databases

- Fast similarity searches and retrieval for high-dimensional vectors

- Consistency guarantees, multi-tenancy, cloud-native, CRUD, logging and recovery, serverless, etc



Source: https://thedataquarry.com/posts/vector-db-1/

Indexing Algorithms in Vector Databases

- Pinecone .... Proprietary composite index
- milvus / zilliz .... Flat, Annoy, IVF, HNSW/RHNSW (Flat/PQ), DiskANN
- Weaviate .... Customized HNSW, HNSW (PQ), DiskANN (in progress...)
- qdrant .... Customized HNSW
- chroma .... HNSW
- LanceDB .... IVF (PQ), DiskANN (in progress...)
- vespa .... HNSW + BM25 hybrid
- Vald .... NGT
- elasticsearch .... Flat (brute force), HNSW
- redis .... Flat (brute force), HNSW
- pgvector .... IVF (Flat), IVF (PQ) in progress...

Common indexes:
HNSW, IVF(PQ)

Source: https://thedataquarry.com/posts/vector-db-1/

Index Algorithms: Big players in the field

- Meta: FAISS (CPU & GPU)
- Google: ScaNN
- Microsoft (Bing team): DiskANN, SPTAG
- Spotify: ANNOY
- Amazon: KNN based on HNSW in OpenSearch
- Baidu: IPDG (Baidu Cloud)
- Alibaba: NSG (Taobao Search Engine)

Source: https://www.youtube.com/watch?v=2o8-dX__EgU&ab_channel=OpenSourceConnections

Different Approaches to ANNS Problem

Graph-based methods

- A navigable graph where each point connects to its neighbor

Product Quantization (PQ)

- Use lossy compression to represent high-dimensional vectors with compact codes

Locality sensitive hashing (no covered in this lecture)

- Use hash functions that map similar points to the same buckets with high probability

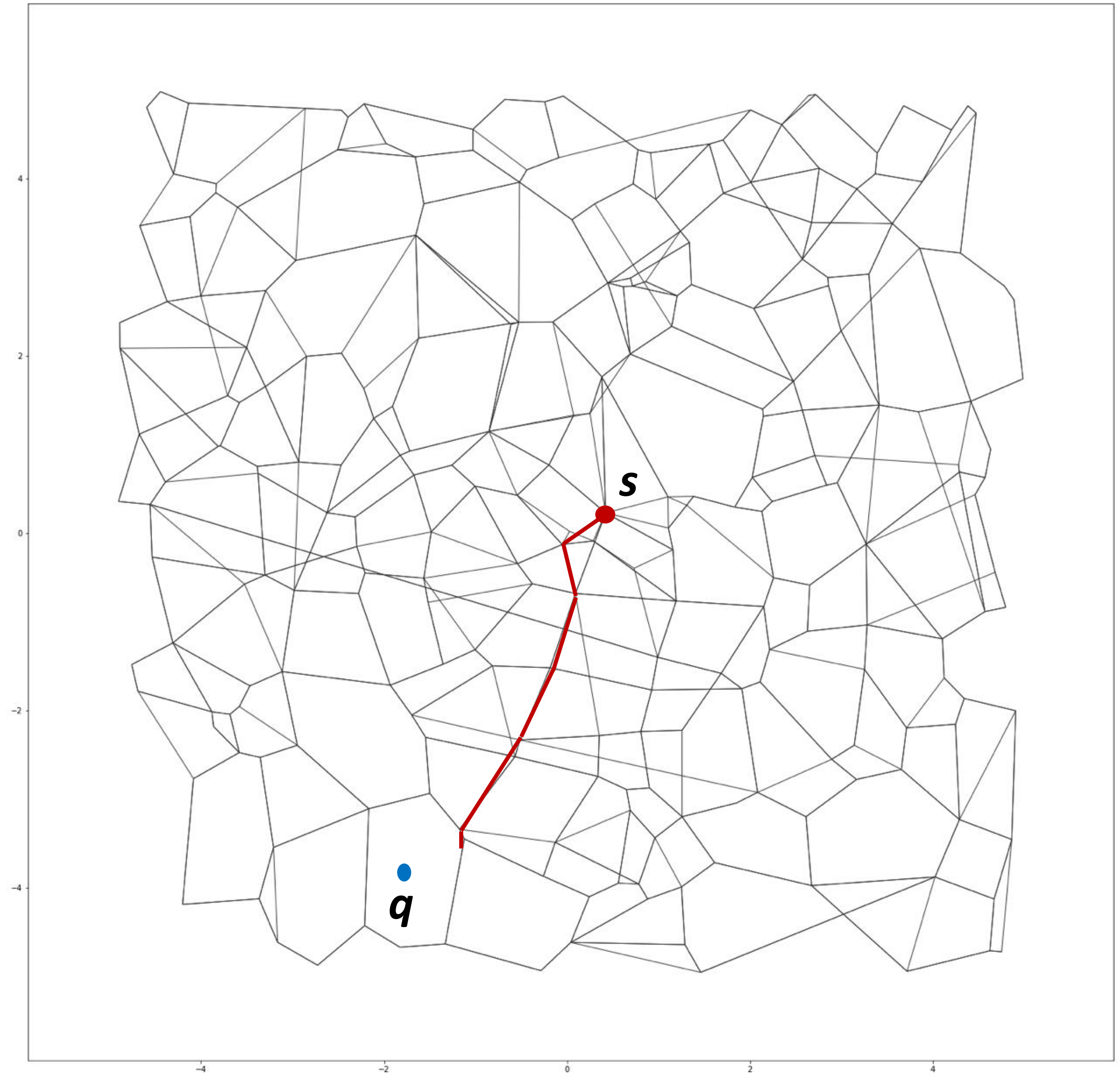
2. Graph-based Methods

Graph-based ANNS: Quick Primer

Offline Stage:
Build a graph over base points, and designate a node s as start.

Online Stage:
For query q, start at a root vertex, traverse the edges as long as distance to q improves

NSG [github.com/ZJULearning/nsg]
HNSW [https://github.com/nmslib/hnswlib]



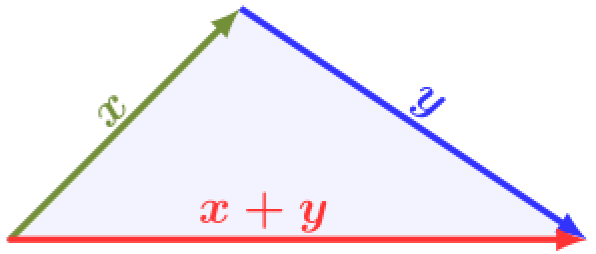
Adapted from DiskANN slides from *Harsha* Simhadri

KNN Graph [WWW'11]

- KNN Graph: for a set of objects V is a directed graph with vertex set V and an edge from each v ∈ V to its K most similar objects in V under a given similarity measure.

- Key intuition: a neighbor of a neighbor is also likely to be a neighbor.

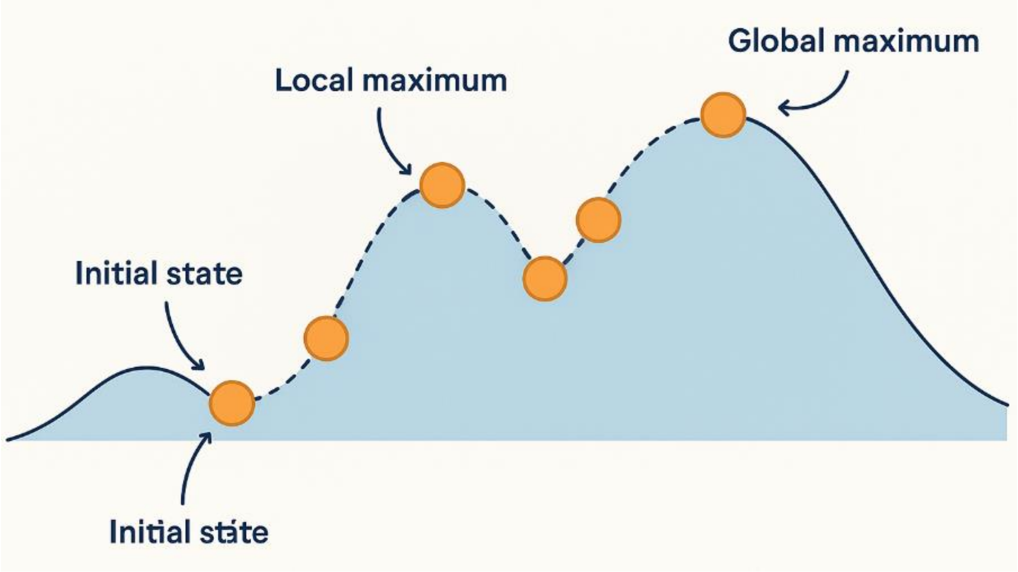
- Triangle inequality:



[Wei2011] Dong, Wei, Charikar Moses, and Kai Li. "Efficient k-nearest neighbor graph construction for generic similarity measures." *Proceedings of the 20th international conference on World wide web*. 2011.

KNN Graph [WWW’11]

- **Search Procedure**: repeatedly move to the closest unvisited neighbor to query (similar to hill climbing), until no closer points can be found.



- **Why this works**: Exploits graph connectivity to quickly traverse from distant regions to the query's neighborhood without exhaustive search.

Image source: https://www.skillcamper.com/blog/hill-climbing-algorithm-in-artificial-intelligence

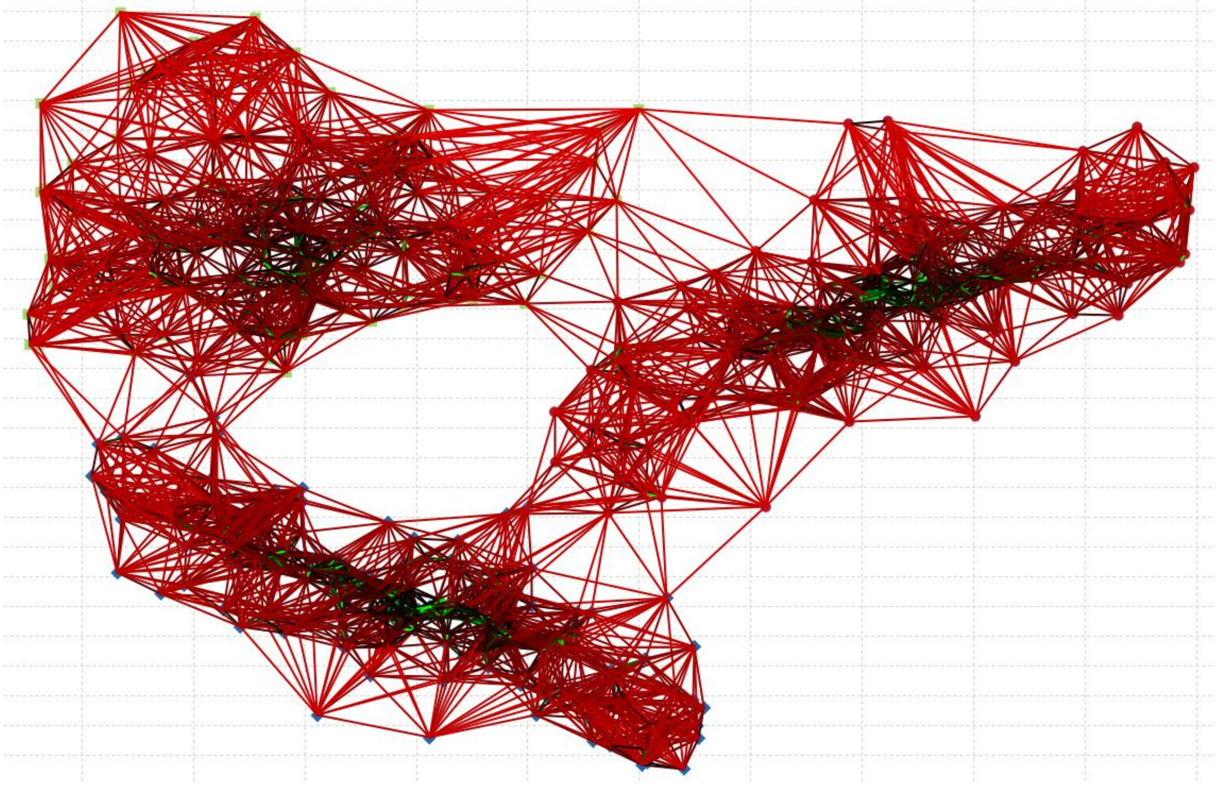
[Wei2011] Dong, Wei, Charikar Moses, and Kai Li. "Efficient k-nearest neighbor graph construction for generic similarity measures." *Proceedings of the 20th international conference on World wide web*. 2011.

KNN Graph [WWW'11]

Challenges:

- To avoid local optima, we need to traverse over thousands of points to find the nearest neighbors of the query point.

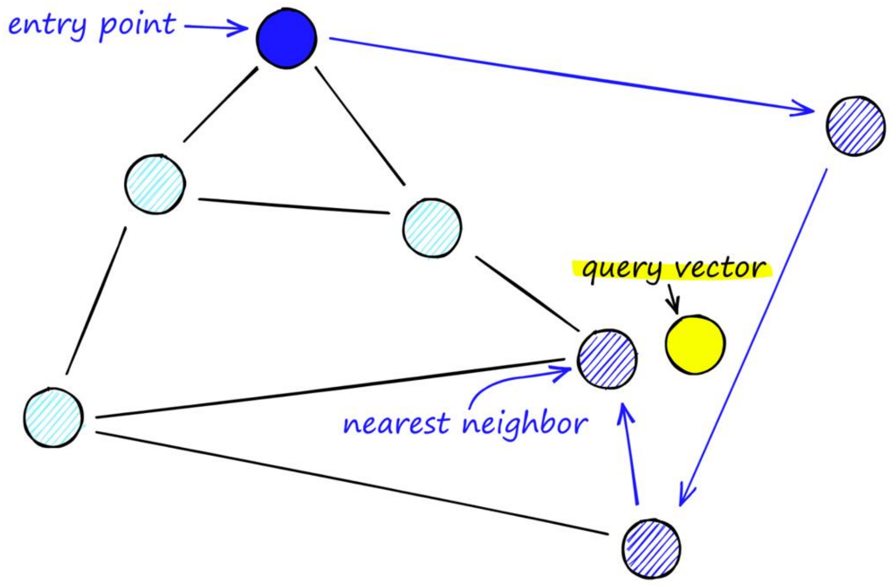
- The size of KNN graph is usually very large and hard to store in memory.



Navigable Small Worlds (NSW)

- A kNN graph that has both long-range and short-range links; inspired by the "small-world" phenomenon, where any two individuals can be connected by a surprisingly short chain of acquaintances.

- NSW adds long-range links using a distribution based on distance: closer nodes are more likely to connect



Long-range links help ensure the search doesn't get stuck in local minima

Source: https://www.pinecone.io/learn/series/faiss/hnsw/

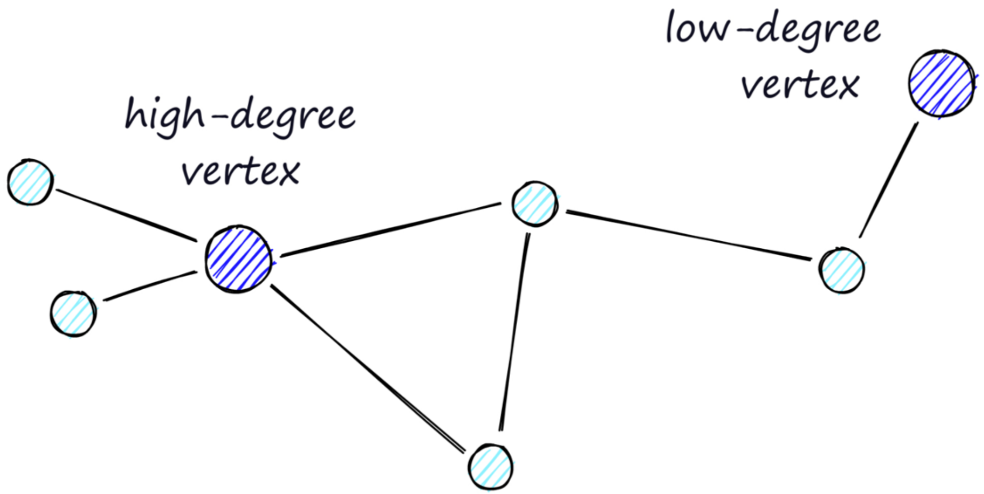
Navigable Small Worlds (NSW)

Can get stuck in local minimal in zoom out phase!

Greedy search procedure:

- Phase 1: Zoom Out (low-degree vertices)
  - Use long-range links to make large jumps across the space
- Phase 2: Zoom in (High-degree vertices)
  - Use dense local connections to refine search

The Degree Dilemma: Increasing the average degree of vertices would increase search complexity – balance between recall and search speed



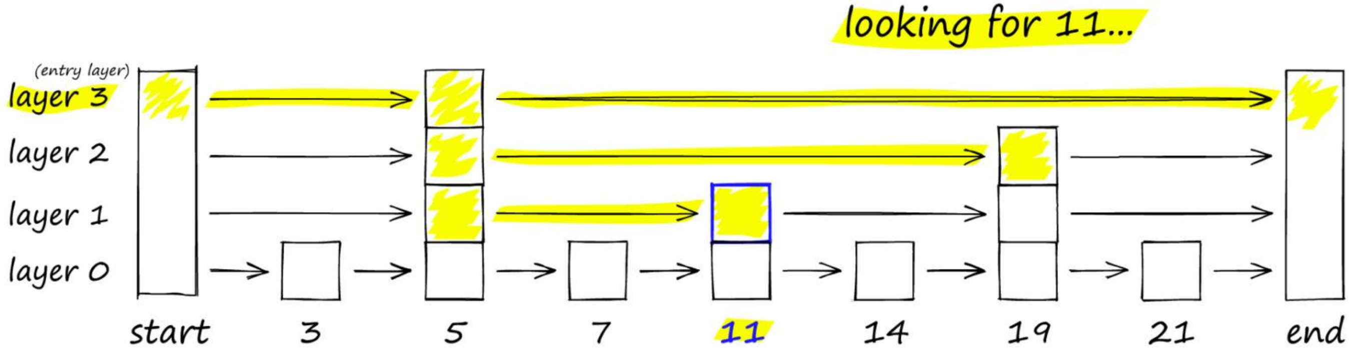
High-degree vertices have *many* links, whereas low-degree vertices have very *few* links.

Source: https://www.pinecone.io/learn/series/faiss/hnsw/

Hierarchical Navigable Small Worlds (HNSW)

Among the top-performing indexes for vector similarity search: fast search speed and good recall

Probability skip list: building several layers of linked lists. On the first layer, we find links that skip many intermediate nodes/vertices. As we move down the layers, the number of 'skips' by each link is decreased.

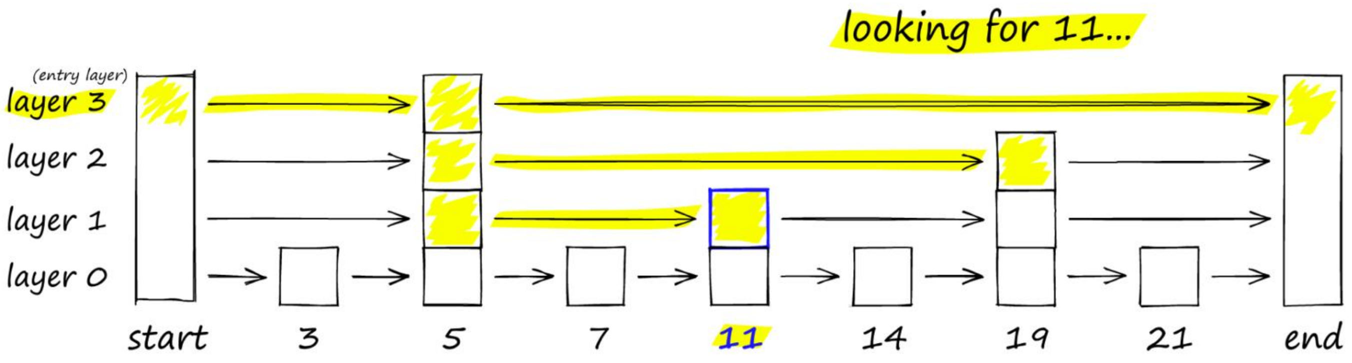


Source: https://www.pinecone.io/learn/series/faiss/hnsw/

Hierarchical Navigable Small Worlds (HNSW)

Search procedure

- Start from the top layer with the longest ’skips’
- If you overshoot, move down to a lower layer

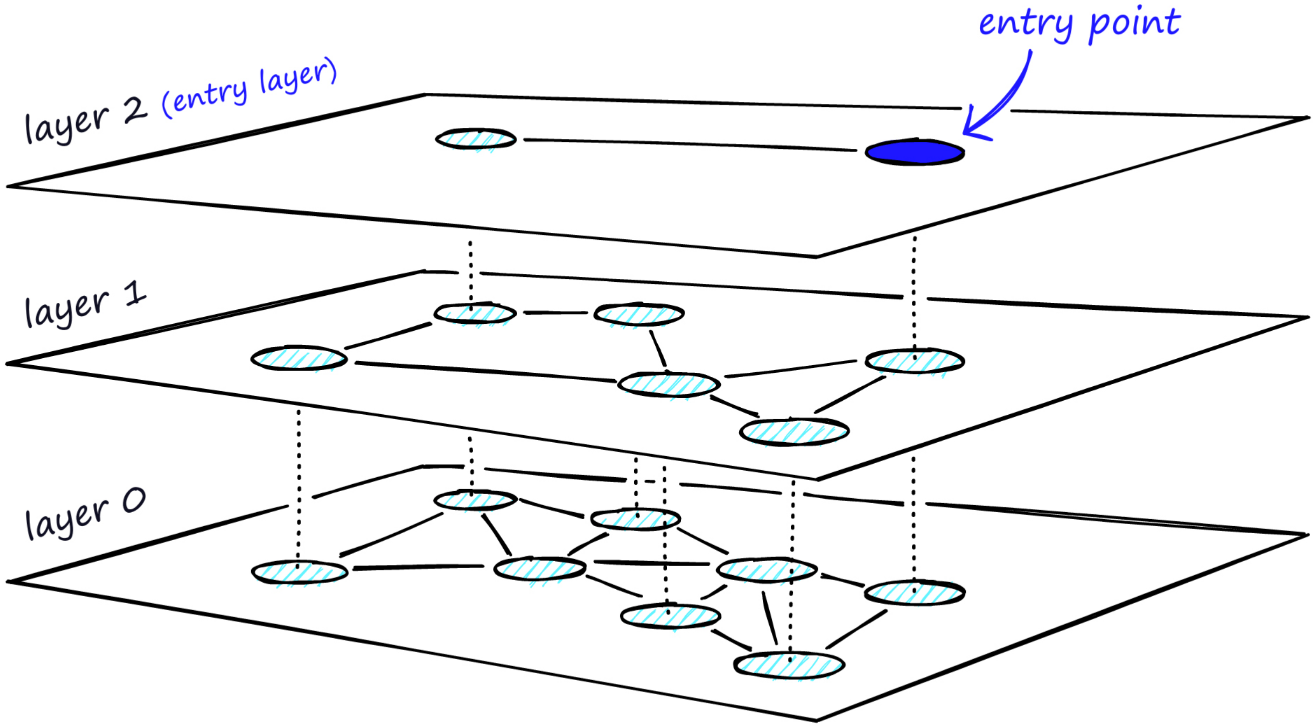


Source: https://www.pinecone.io/learn/series/faiss/hnsw/

Hierarchical Navigable Small Worlds (HNSW)

Main idea: Combine skip list with NSW

- Top layers: few nodes, long links
- Bottom layer: all nodes, short links
- Middle layer: gradually increase density



Separation of concerns:

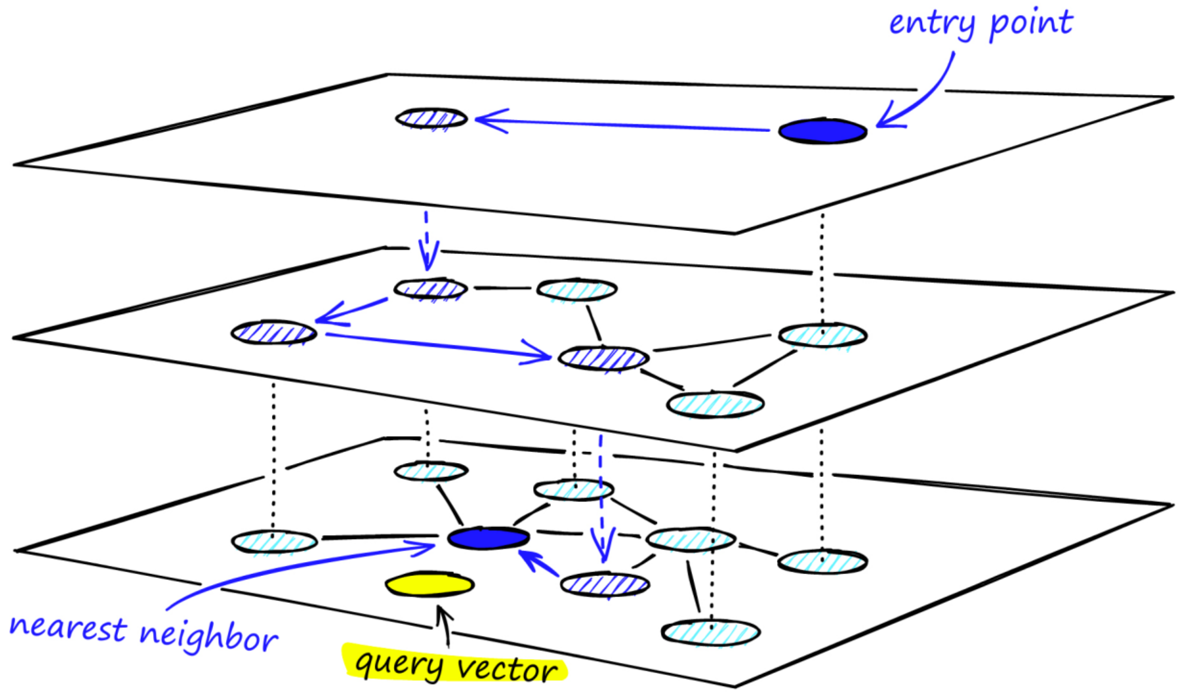
- Top layers are optimized for long-range navigation (zoom out)
- Bottom layers are optimized for accurate local search (zoom in)

Source: https://www.pinecone.io/learn/series/faiss/hnsw/

Hierarchical Navigable Small Worlds (HNSW)

Search procedure

- Enter from top layer:
  - A point in the top layer has few edges in the top layer, but it also has edges in all lower layers
  - **Total degree** across all layers is high, even though degree within top layer is low
- Upon finding local minimum, descend to a lower layer and repeat the search

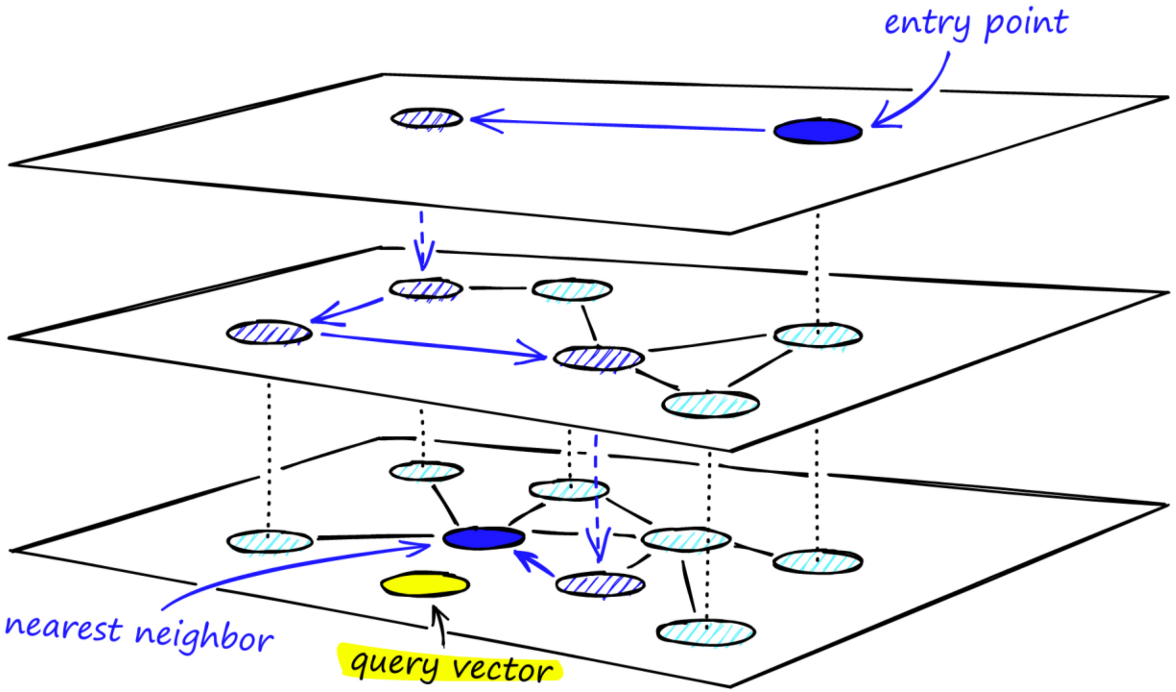


Source: https://www.pinecone.io/learn/series/faiss/hnsw/

Hierarchical Navigable Small Worlds (HNSW)

Index Construction

- Step1: assign layer level
  - Randomly determine maximum layer ℓ for the new point
  - $P(layer = \ell) \propto e^{-\ell}$
- Step2: Insert and connect at each layer
- Step3: prune connections (optional)



Source: https://www.pinecone.io/learn/series/faiss/hnsw/

Hierarchical Navigable Small Worlds (HNSW)

HNSW is an in-memory index:

- Entire graph structure and vectors stored in RAM
- For each node, we need to store:
  - vector data (used for distance computation)
  - adjacency list (neighbor list for each layer 0 to ℓ)

What if server doesn't have enough memory:

- PQ: Compress vectors to save space (discussed next)
- Partition and distribute across machines (nontrivial due to network communication)
- DiskANN [NeurIPS'19], SPANN [NeurIPS'21]: memory-SSD hybrid solution

3. Product Quantization

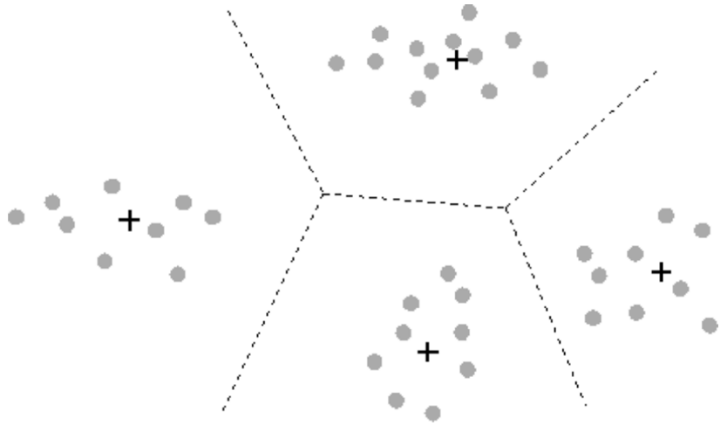
Product Quantization

Winner in BigANN Competition @ NeurIPS' 21; a technique for compression high-dimensional vectors, therefore speeding up the similarity search.

Popular implementation: Meta's faiss library

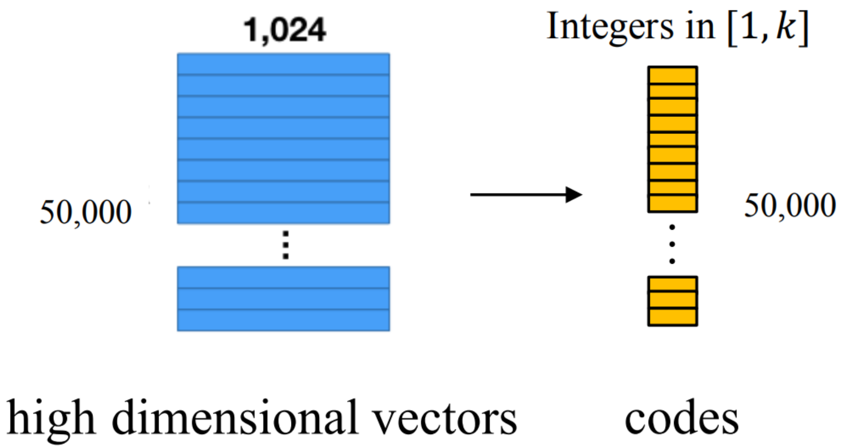
Vector Quantization: use centroids to represent vectors in clusters.

- distance(query, vector) ~ distance(query, centroid)



Vector Quantization

- Map the original dataset by a vector quantizer with k centroids using k-means
- Store only the centroid ID (integer code) instead of full vector
- Codebook: set of k centroids (learned from data)

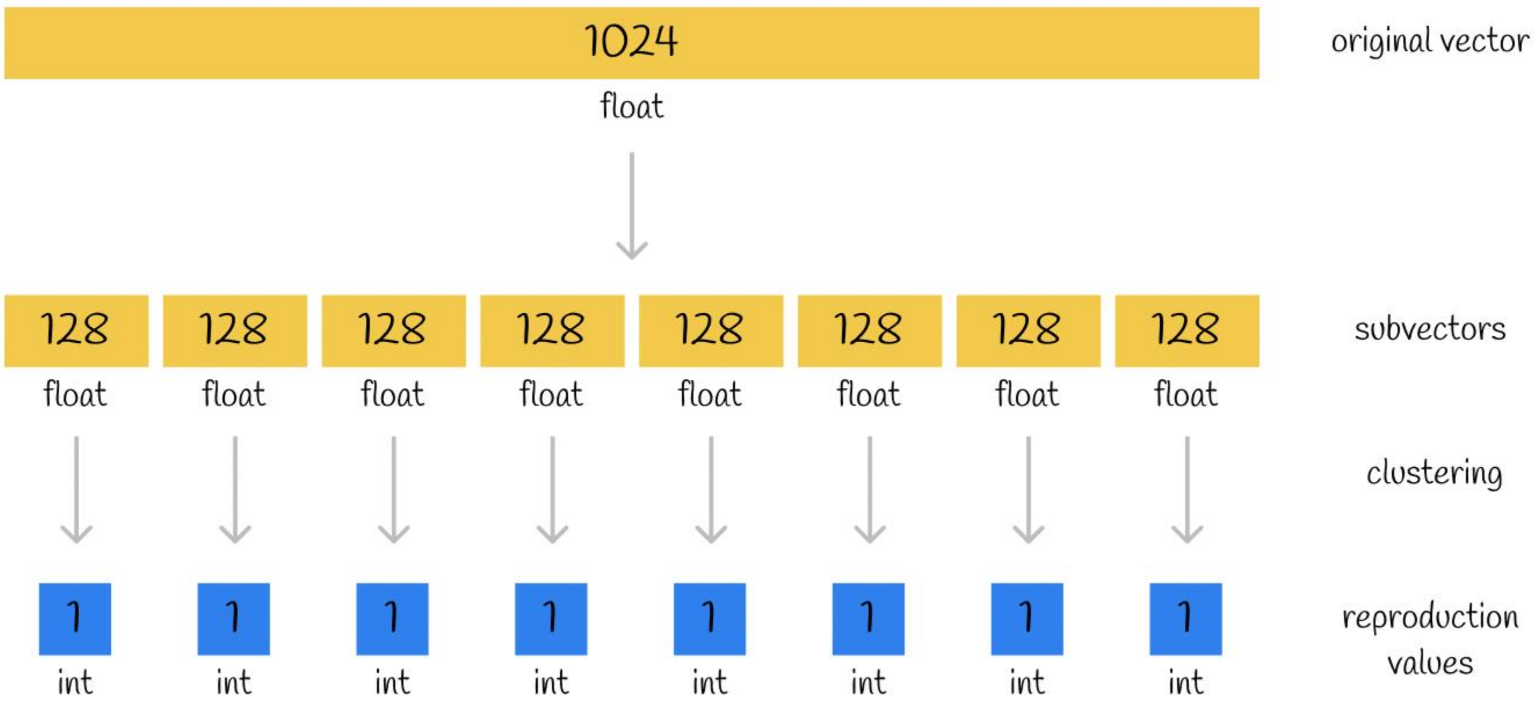


Problem: need a large number of clusters to distinguish vectors

- e.g., a quantizer producing 64-bit code contains k = 2^{64} centroids

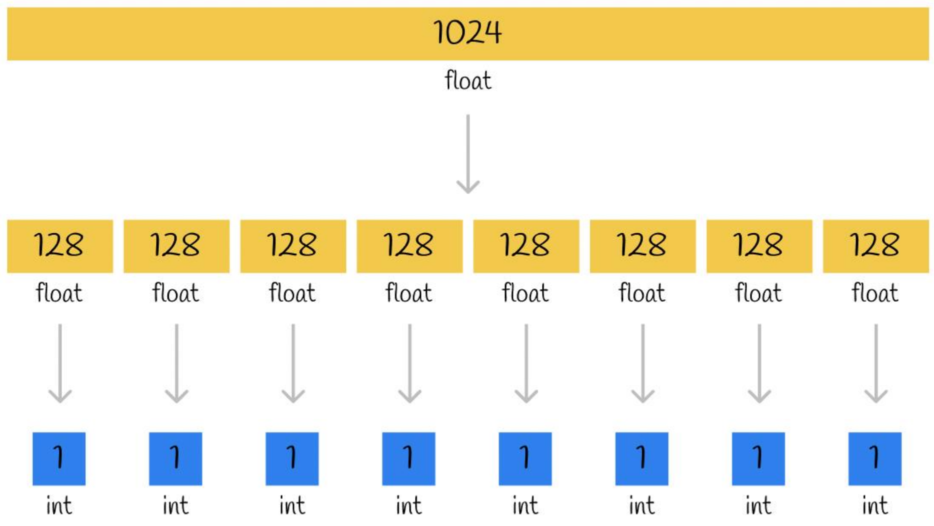
Product Quantization

- Split a high-dimensional vector into equally sized subvectors
- Assigning each of these subvectors to its nearest centroid
- Replacing these centroid values with unique IDs — each ID represents a centroid



https://towardsdatascience.com/similarity-search-product-quantization-b2a1a6397701

Product Quantization



Benefit: Produce a large set of centroids from several small sets of centroids

Suppose we are using 32 bits for each compressed vector

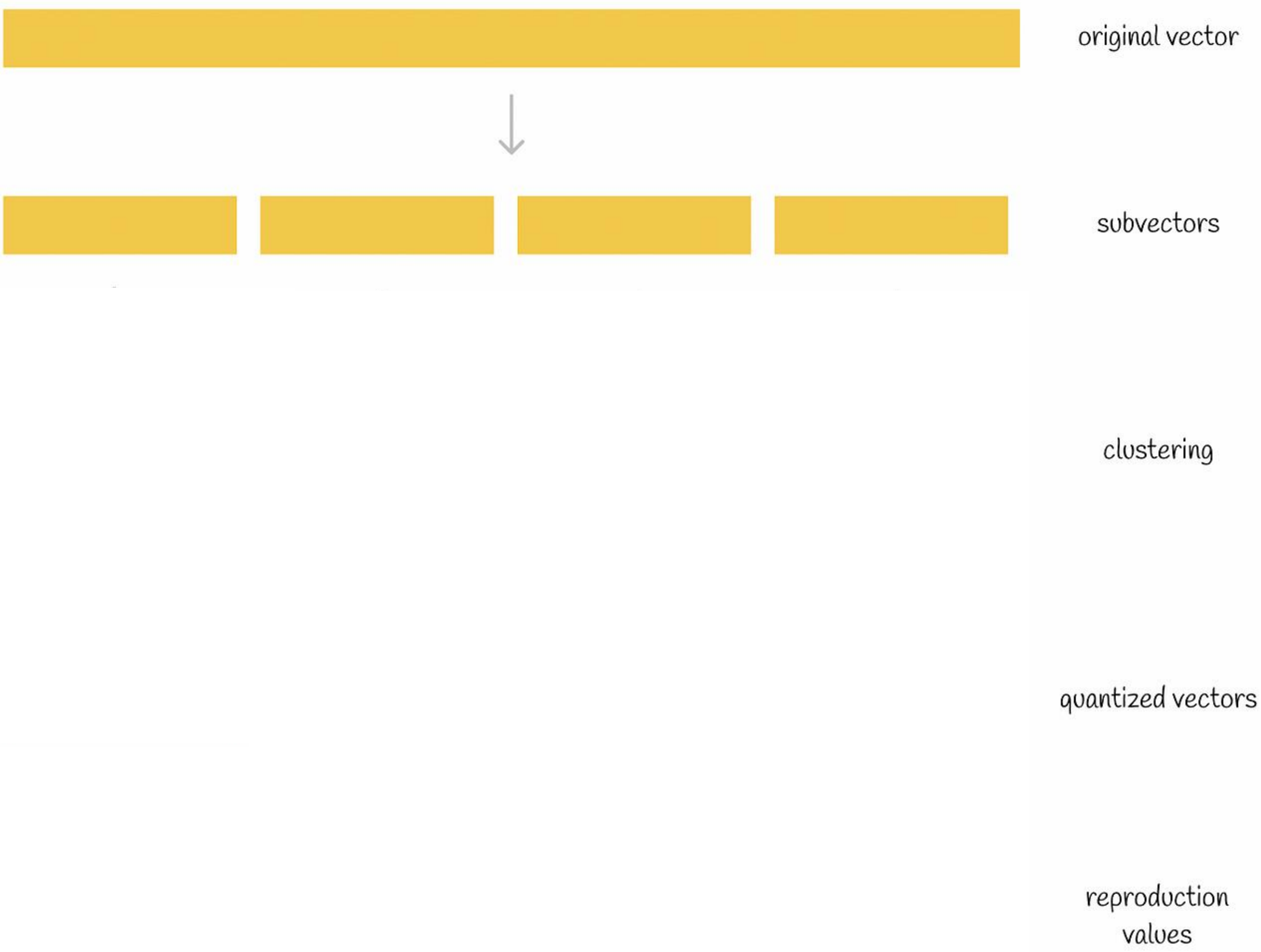
- Vector quantization:
  - $k = 2^{32}$ total centroids
  - Total centroids: $k = 2^{32} = 4{,}294{,}967{,}296$

- Product quantization:
  - $m = 4$ subquantizer
  - $k^* = 2^8$ centroids for each subquantizer
  - Total centroids: $m \cdot k^* = 1024$

$$k = (k^*)^m$$

https://towardsdatascience.com/similarity-search-product-quantization-b2a1a6397701

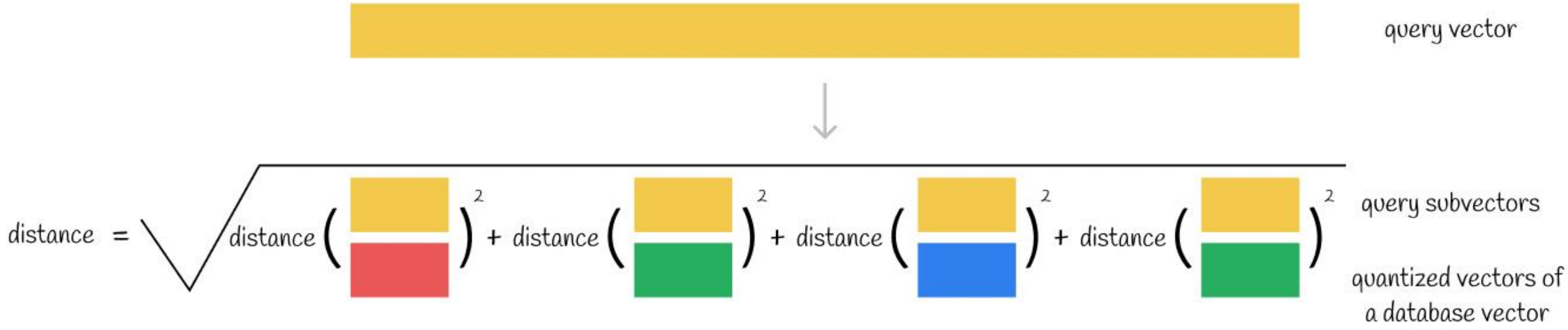
Quantization



Each subvector space has its own set of clusters

https://towardsdatascience.com/similarity-search-product-quantization-b2a1a6397701

Computing Distances with Quantized Codes



Asymmetric distance computation: The database vector y is represented by $q(y)$, but the query x is NOT encoded.

$$\tilde{d}(x,y) = \sqrt{\sum_j d(u_j(x), \boldsymbol{q_j}(u_j(y)))^2}$$

https://towardsdatascience.com/similarity-search-product-quantization-b2a1a6397701

Using PQ in Indexes

PQ is just a lossy compression mechanism to reduce the memory footprint of vector data

During ANN search, still need an index to avoid exhaustive search

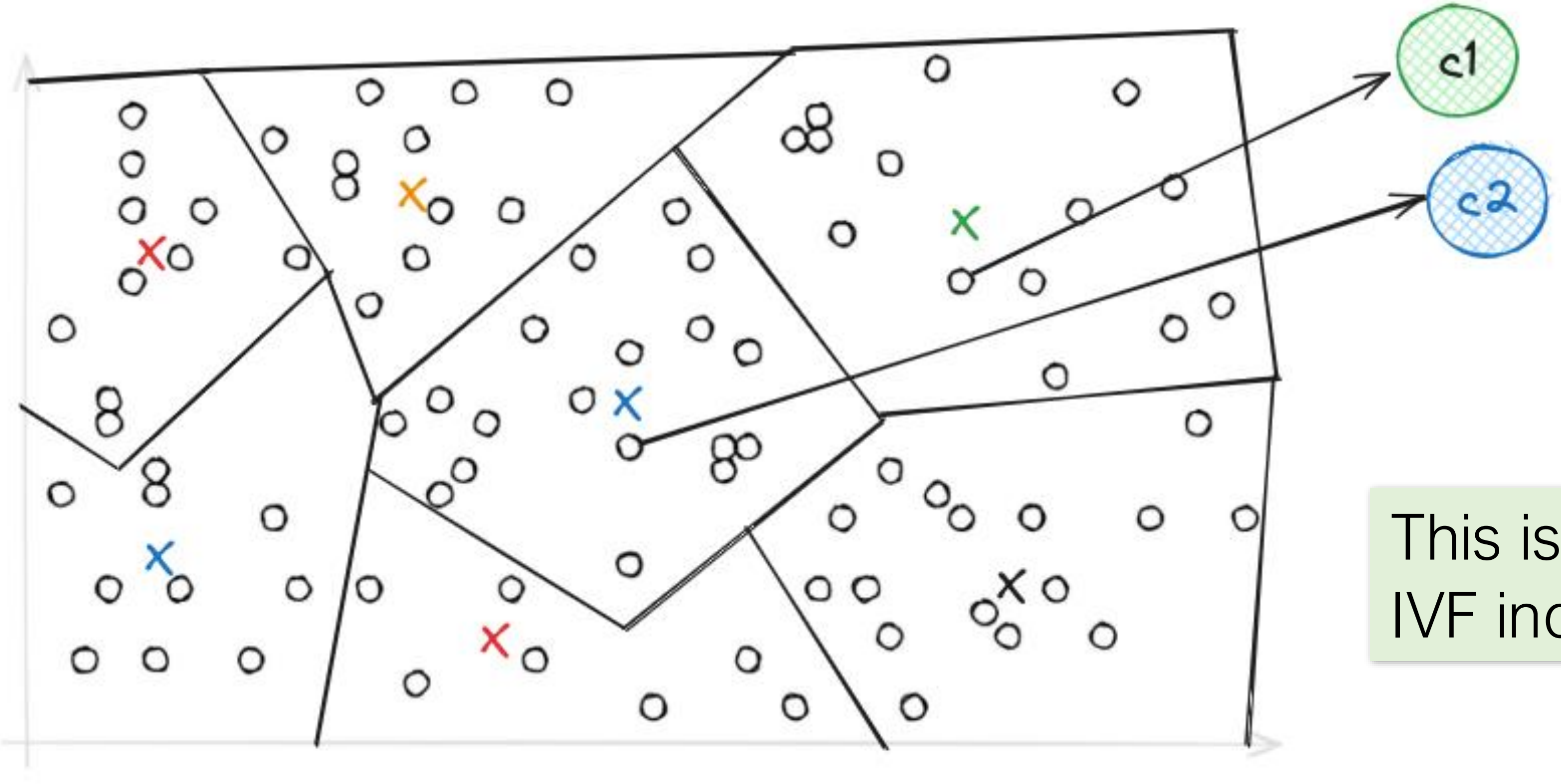
We will explore two examples:

- IVF-PQ: inverted index + PQ
- DiskANN: graph-based method that uses PQ for the in-memory component

IVF-PQ: Inverted File Index

Small memory footprint, but lower recall due to lossy compression

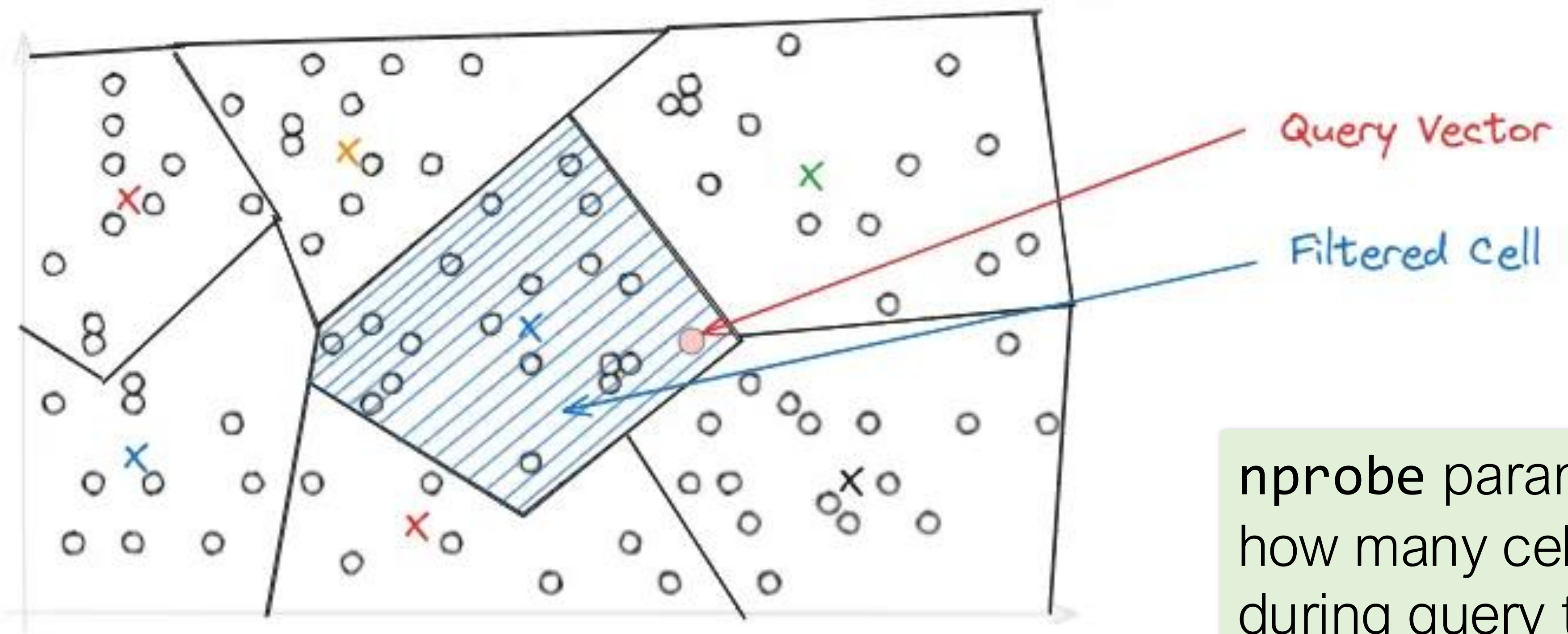
Assign all vectors to Voronoi cells (e.g., via K-Means clustering)



This is the coarse-grained IVF index

IVF-PQ: Inverted File Index

During search, only check nearby cells



nprobe parameter controls how many cells to search during query time => controls search recall vs latency

https://lancedb.github.io/lancedb/concepts/index_ivfpq/#product-quantization

IVF-PQ: Index Construction

Step 1: Coarse Partitioning with IVF

- Use K-Means to partition the datase
- Assigns each vector to its nearest centroid

Step 2: Compression with PQ

- For each vector in the database, calculate the **residual**:
  - Residual = Original Vector - Assigned Centroid
- Compress the residuals using PQ
  - Result: Each vector represented by (centroid ID, PQ code)

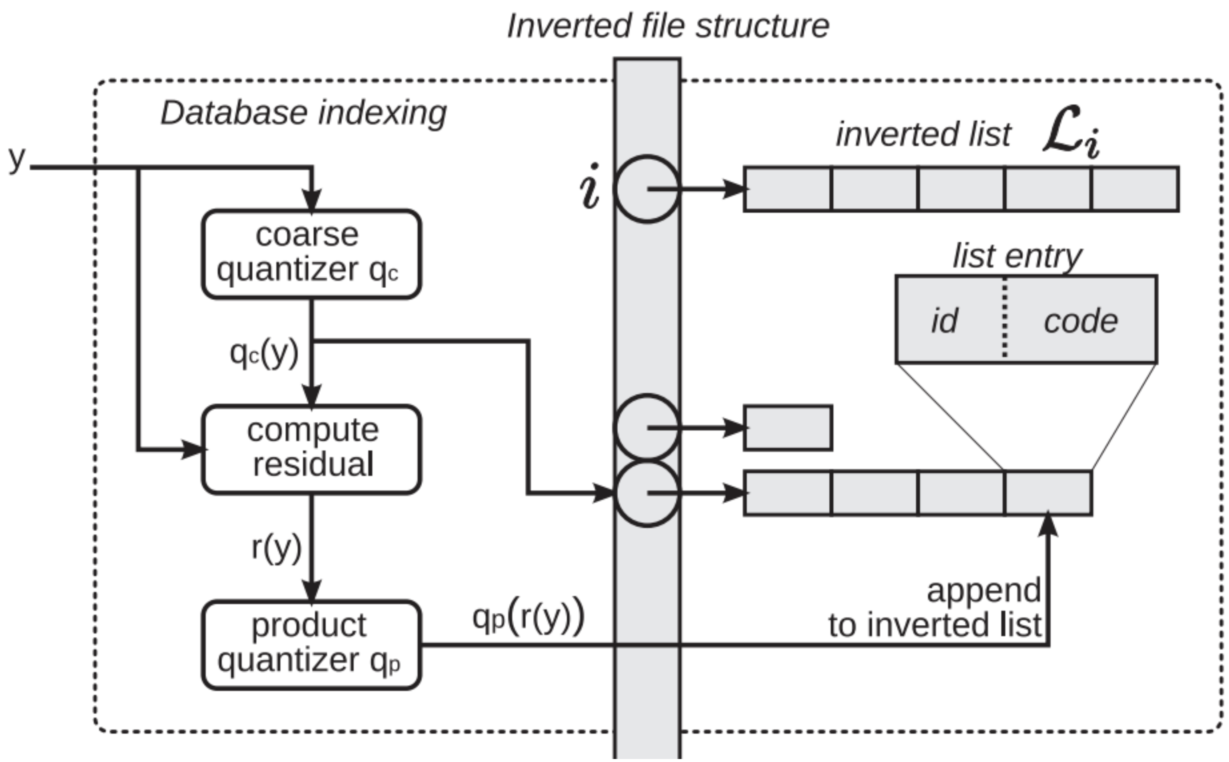
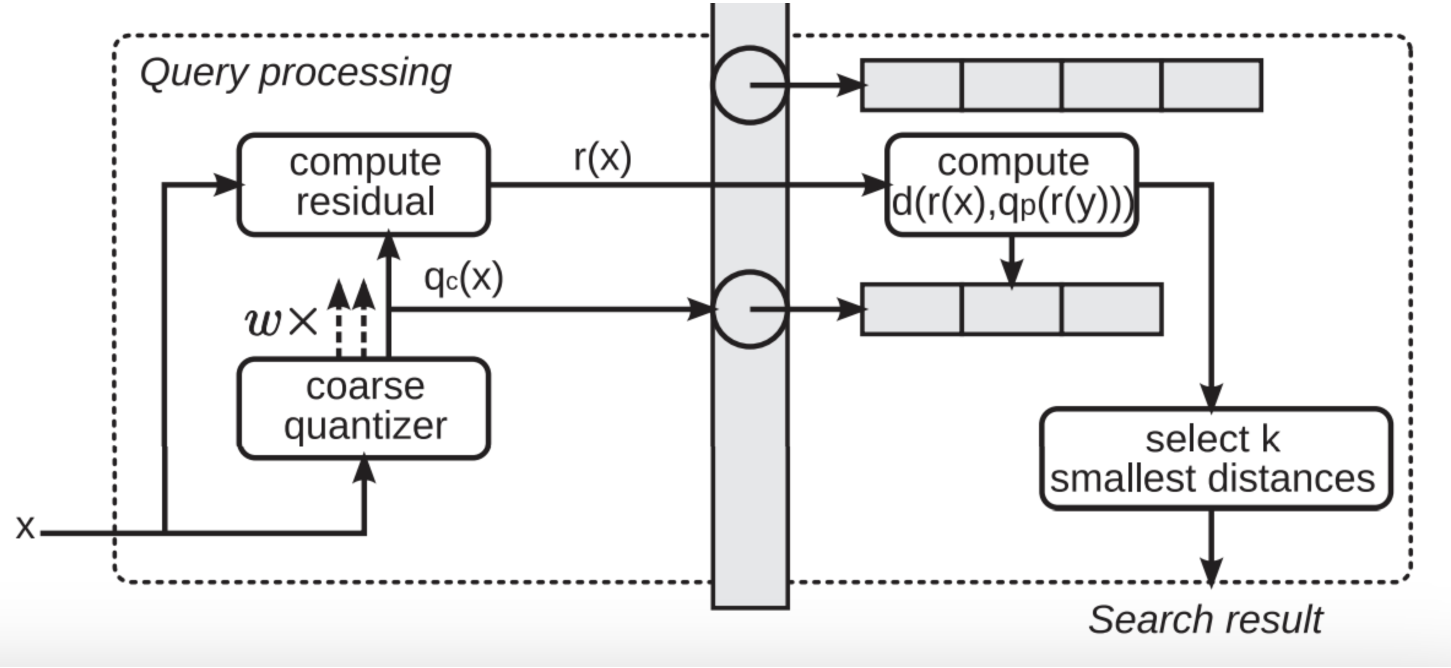


Image source: Product Quantization for Nearest Neighbor Search

IVF-PQ: Search



Step 1: Coarse Search (IVF)

- Select the *nprobe* nearest centroids to query vectpr

Step 2: Fine Search with PQ

- For each selected centroid, compute query residual:
  - Query Residual = Query – Centroid
- Estimate distances between query residual and database residuals and rank candidates by distance

Image source: Product Quantization for Nearest Neighbor Search

DiskANN [NeurIPS'19]: Memory-SSD

In Memory:

- Compressed vectors (PQ codes) for ALL points in dataset
- First few levels of graph

On SSD:

- For each node v:
  - Full-precision vector of v + adjacency list of v
  - Co-located for efficient single-read access (1 I/O for each node)

In full precision, 1B points in 100 dimensions would consume 400 GB RAM, but we can achieve very good results by storing them as compressed coordinates with ~32GB

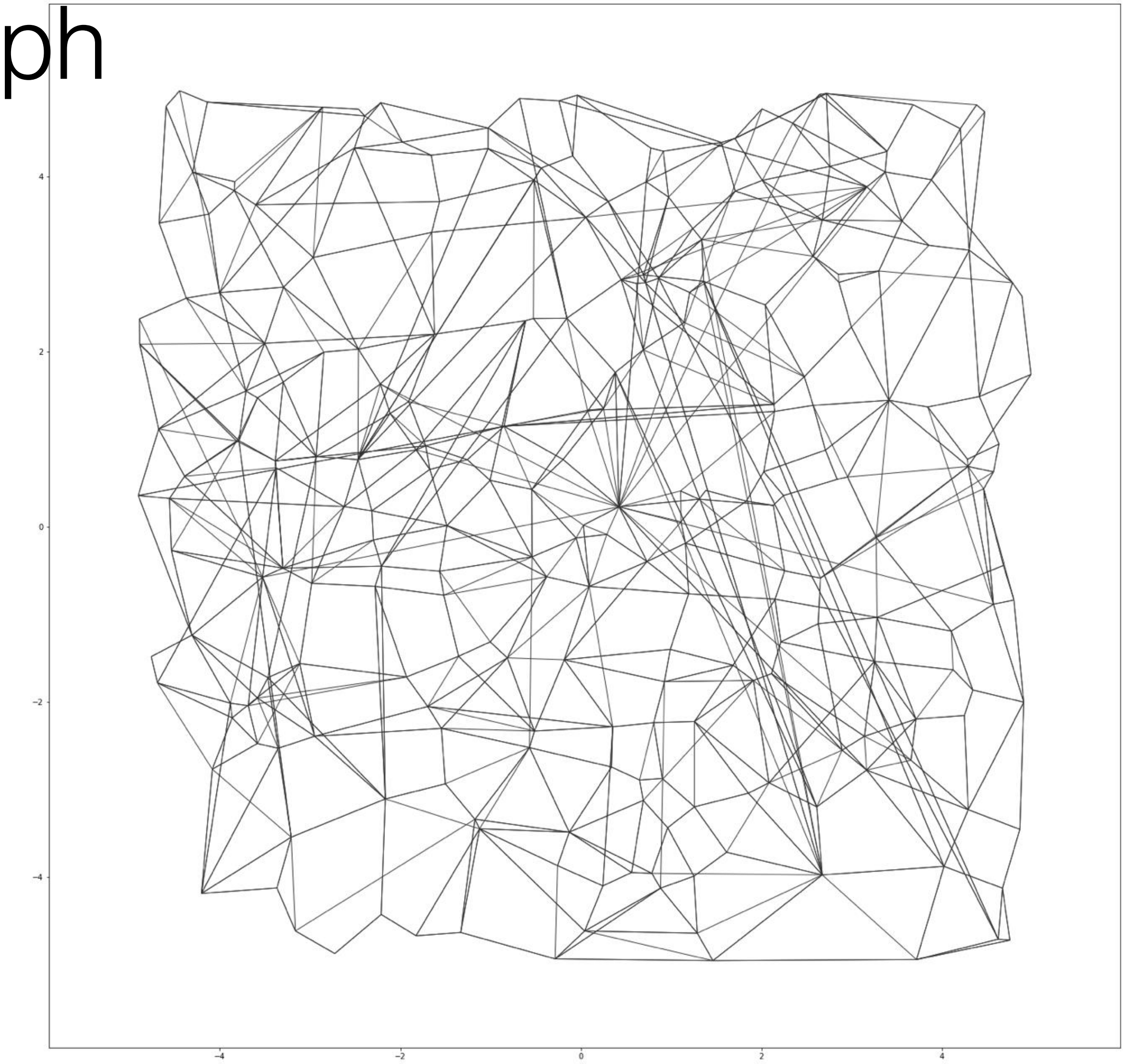
Source: DiskANN NeurIPS slides

DiskANN: Vamana Graph

Graph construction algorithm used by DiskANN, optimized for

- Small graph diameter than NSG, HNSW: fewer disk reads
- Degree bounds: each node's data can fit into one page

Graph initialized with random connections, and can quickly converge



Source: DiskANN NeurIPS slides