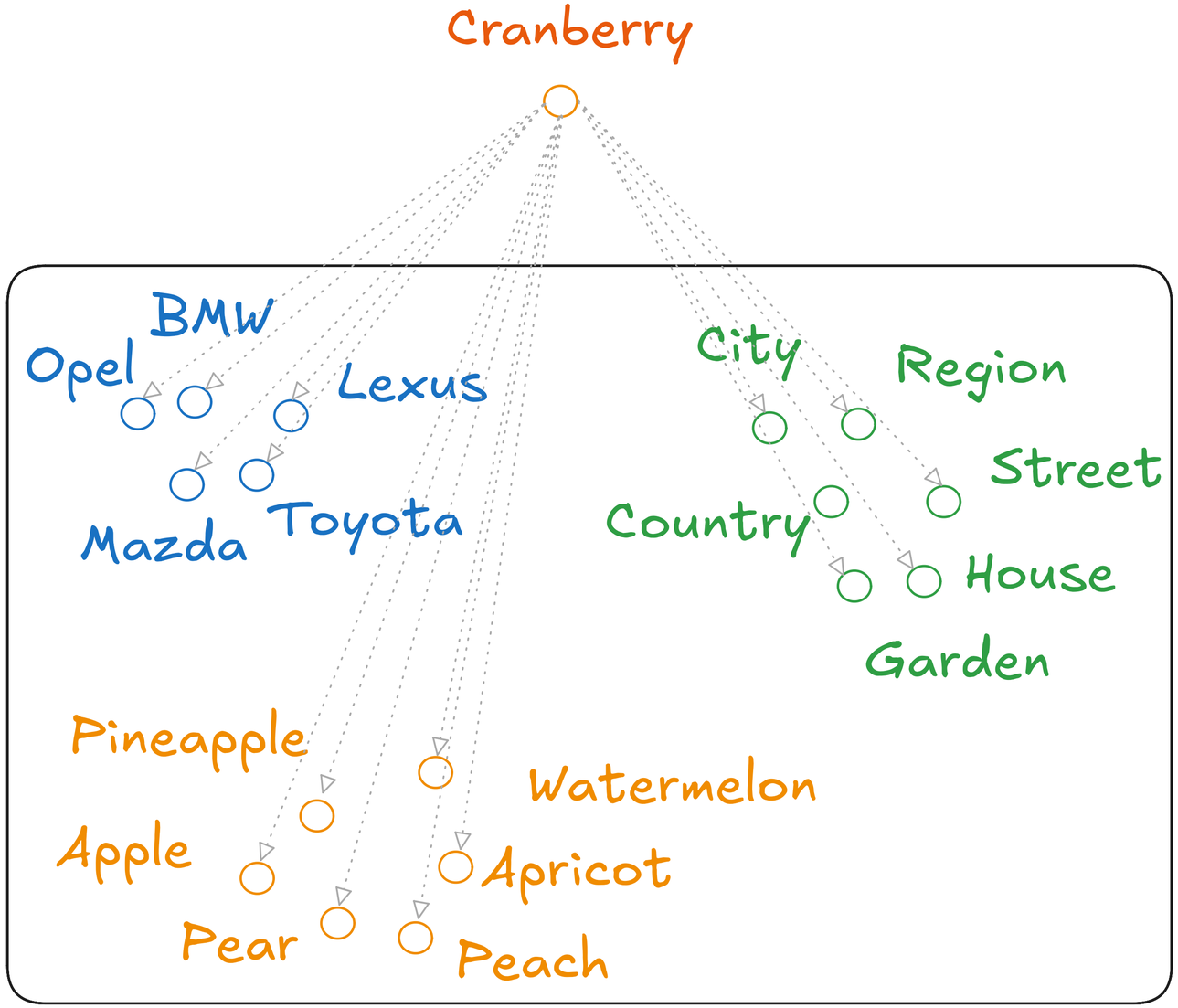
Approximate nearest neighbour search (ANN)

George Panchuk
HSE AI Fall 2025

github.com/joein/vector-search-course

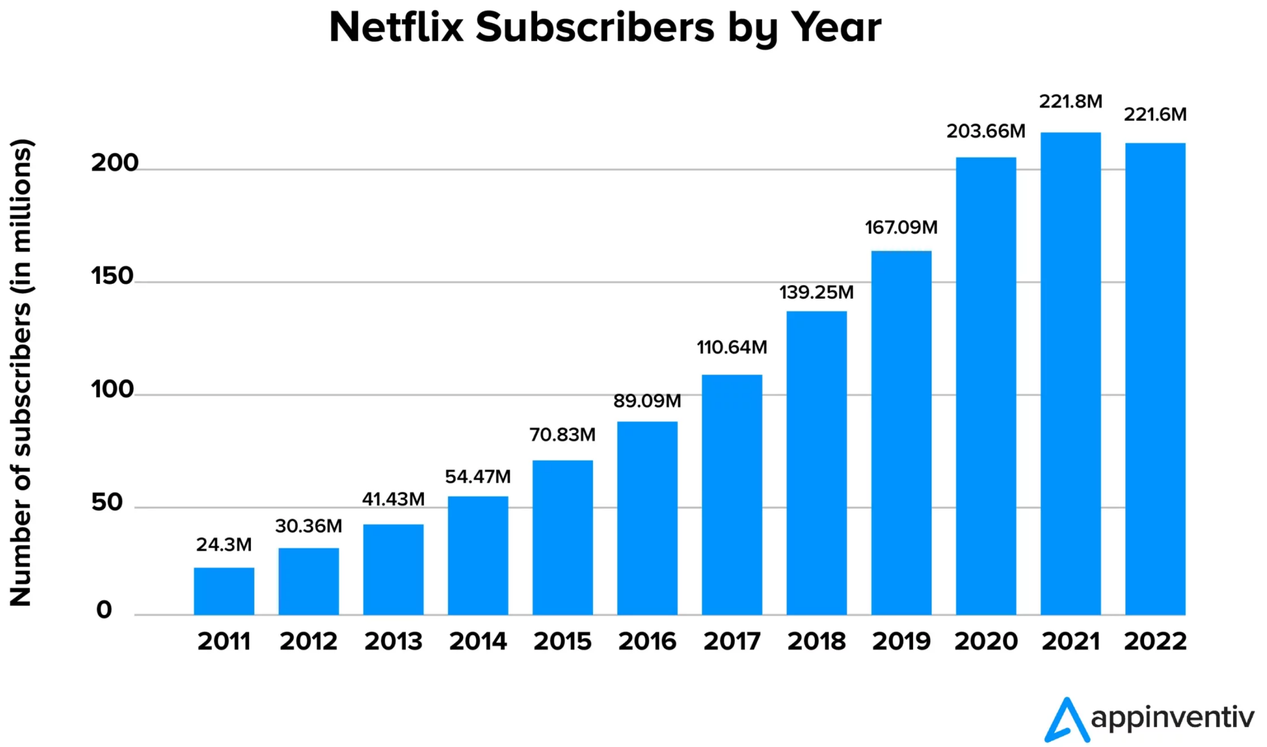
Recap: Brute-force

- Compares query vector to all the vectors in a collection
- Computes exact similarity
- Guarantees 100% vector recall
- $O(N \cdot D)$ complexity
- Stores only vectors, no index overhead
- Performs well on small datasets (20-100k vectors), slow and costly for large
- Can be used to benchmark ANN algorithms



The challenge of scalable vector search

- Modern systems use millions → billions of embeddings
- Real-time search requires low latency (e.g., 50ms)
- Hardware costs grow with dataset size → too expensive
- Need methods that avoid touching every vector on every query



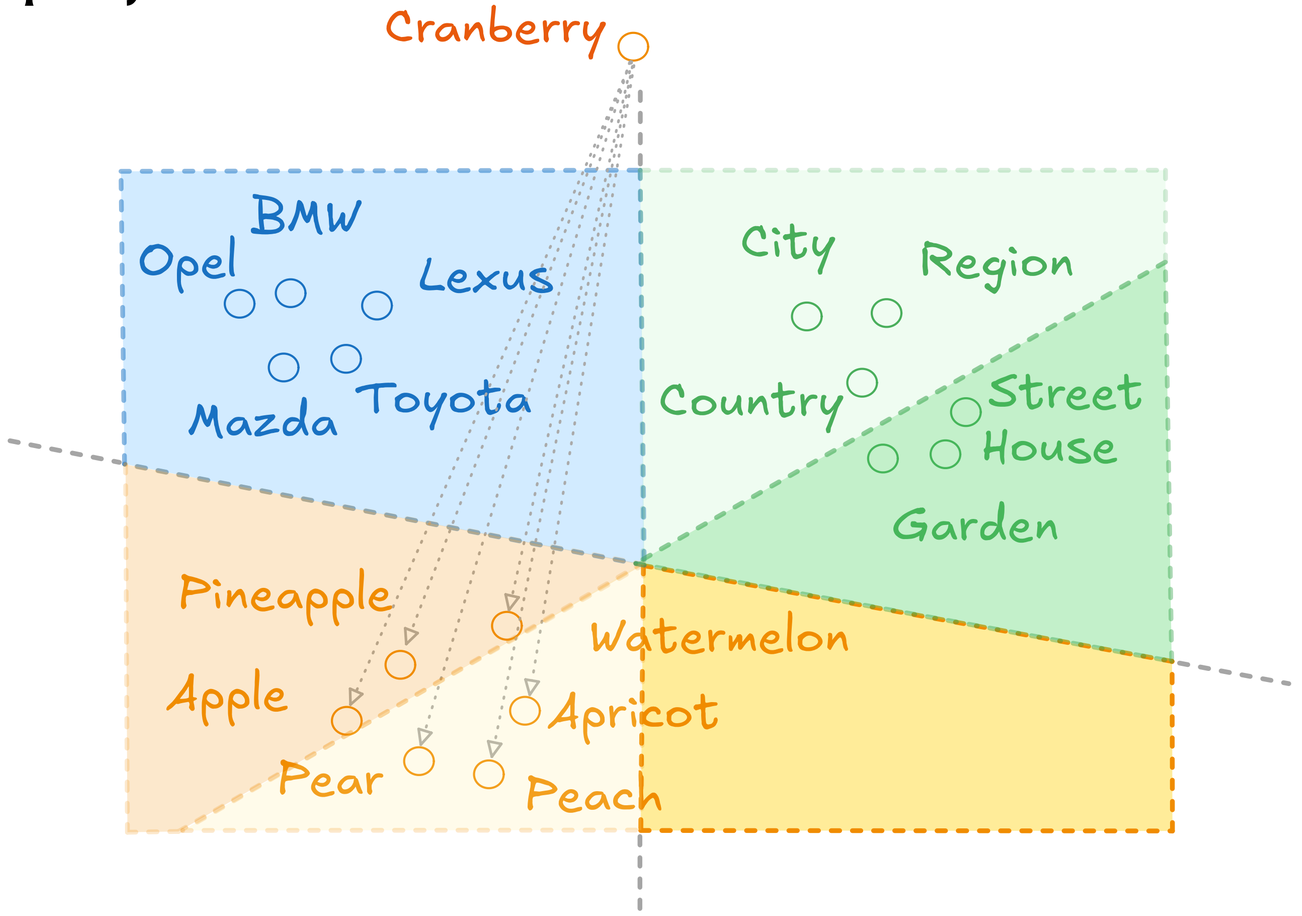
https://appinventiv.com/blog/netflix-statistics-facts/

Approximate Nearest Neighbour (ANN)

Introduction

ANN - is a family of algorithms that accelerate nearest neighbour search by allowing controlled error in exchange for sublinear query time.

- Search only a subset of data (might miss the absolute closest neighbour)
- Achieves sublinear query time in practice
- Requires an upfront index-building phase
- Approximation is tuneable, allowing trade-offs between speed, memory and accuracy



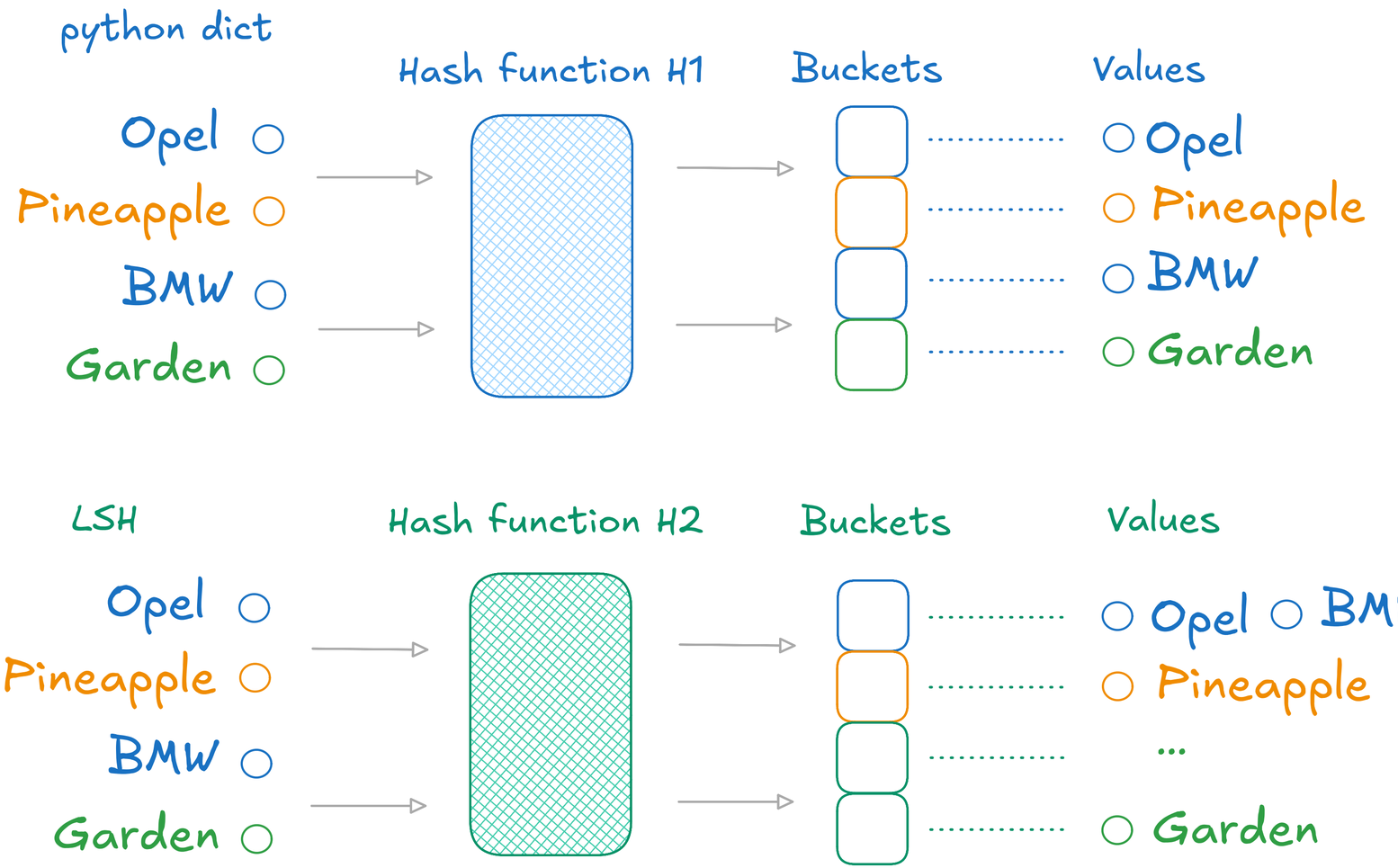
ANN

Variations

- Hash-based methods
  Idea: map vectors to buckets via probabilistic hashing
  Examples: LSH, SimHash
- Tree-based / Space partitioning methods
  Idea: Partition space hierarchically
  Examples: KD-Tree, Ball-Tree, Randomised KD-tree, ANNOY
- Quantization / IVF-based methods
  Idea: compress vectors via centroids / codes; search within clusters
  Examples: IVF, PQ, IVFPQ
- Graph-based methods
  Idea: Build a proximity graph, traverse for queries
  Examples: NSW, HNSW, DiskANN, Vamana

Locality Sensitive Hashing (LSH)

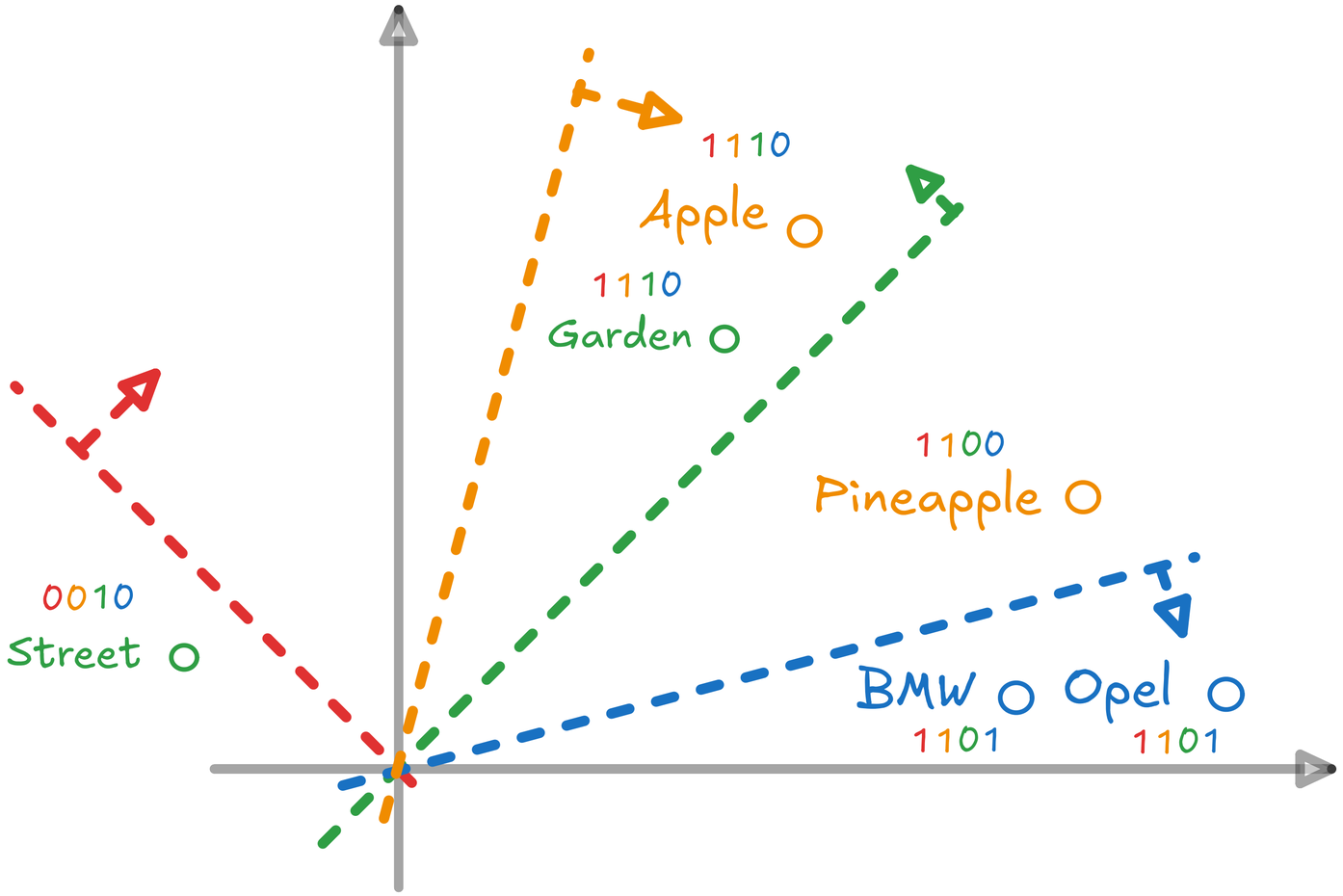
- Distribute vectors among buckets created by hash functions
- Similar vectors should fall into the same bucket
- Can give probabilistic guarantees about collisions of similar items
- To index: compute hash functions on the object and find its buckets
- To query: compute hash functions, find the right buckets, compare vectors from those buckets to the query
- Generate L different tables to lower false negatives



Hashing based methods

LSH with random projections (SimHash)

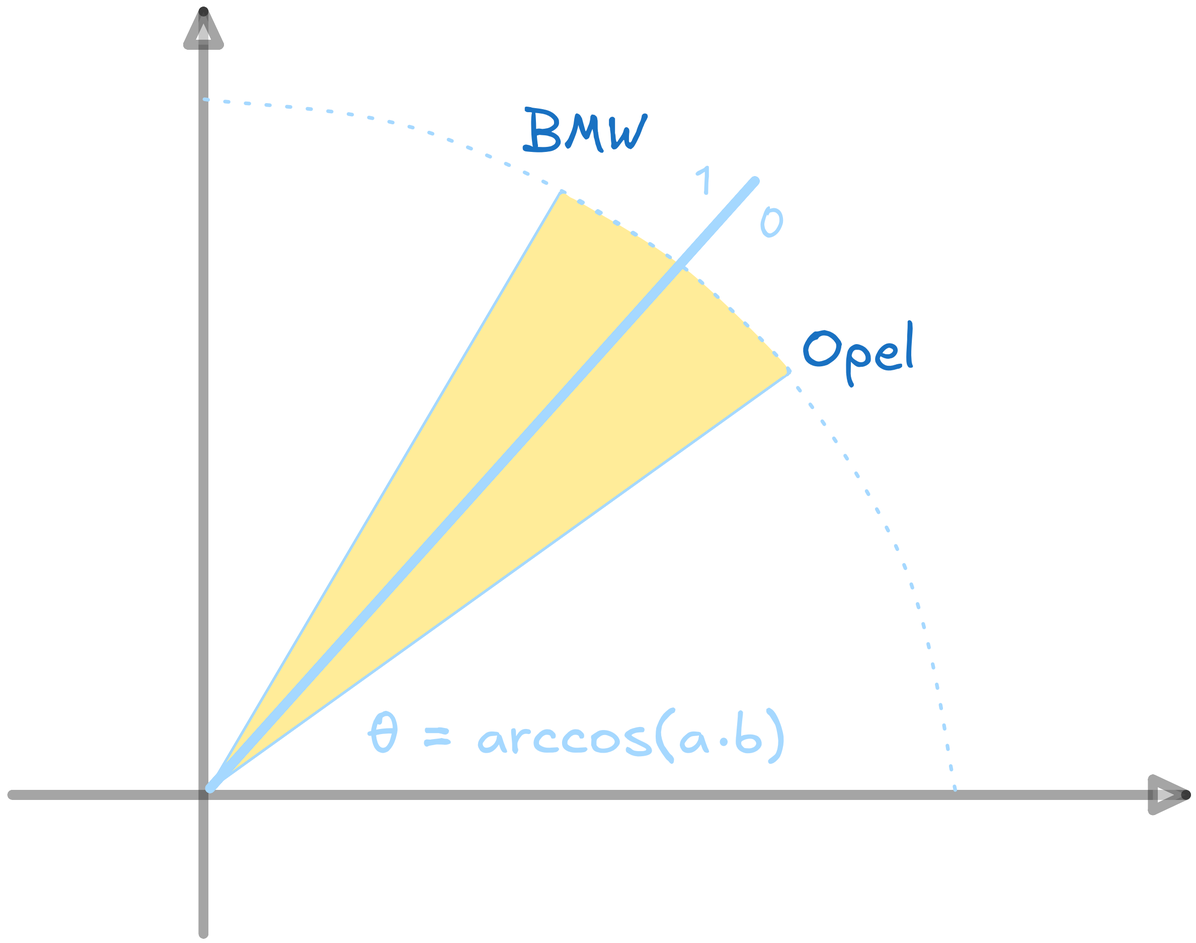
- Use projection on random hyperplanes as hash functions (a set of hyperplanes is a table)
- If dot product between a vector and a random hyperplane $\geq 0 \rightarrow$ assign 1, otherwise assign 0
- *nbits* - is a number of hyperplanes (length of a hash-code)
- Use Hamming distance to find similar points
- Rerank with the original vectors
- Generate L tables to improve recall at a cost of speed and memory



Hashing based methods

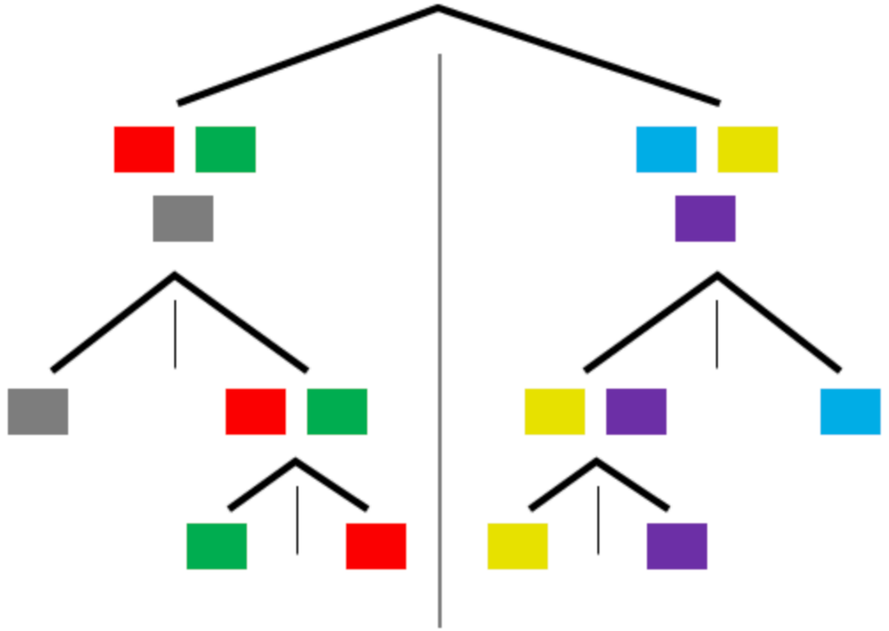
SimHash

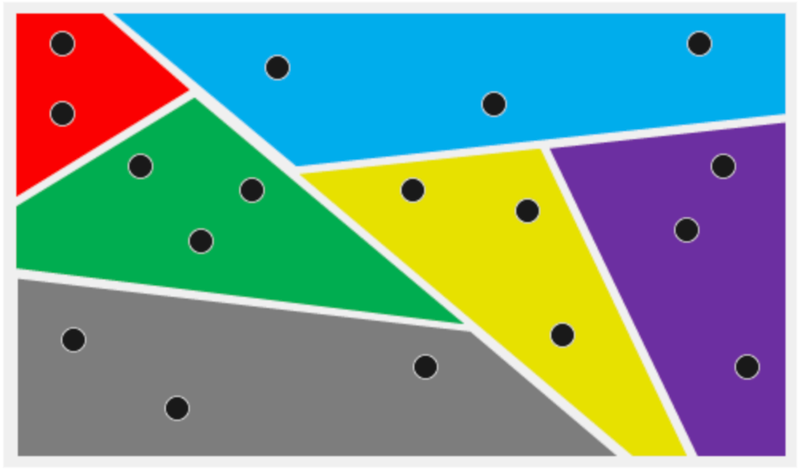
- Approximates cosine similarity
  $Pr[h_i(x) = h_i(y)] = 1 - \frac{\theta(x, y)}{\pi}$
  , where $\theta(x, y)$ is the angle between vectors
- Typically, hash length
  $k = 64..128$
- Better for high-dimensional sparse vectors
- Other algorithms usually outperform in recall and speed



Tree-based methods

- Organize vectors hierarchically to speed up nearest neighbour search
- Each node represents a subset of points
- Traverse tree and prune irrelevant branches for querying
- Works best for low- to medium-dimensional vectors
- kd-tree, Ball tree, ANNOY, etc

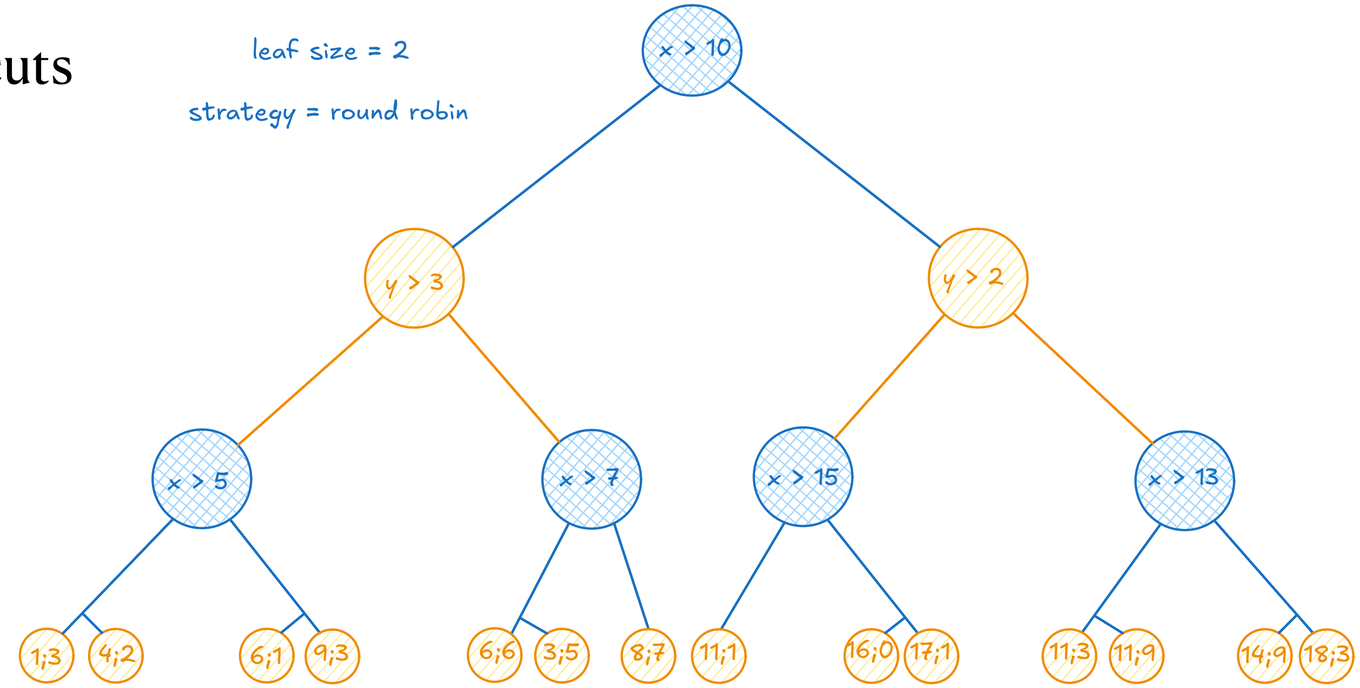




Tree-based methods

kd-tree

- Recursively splits points using axis-aligned cuts (median along chosen axis)
- Produces a balanced binary tree of nested bounding boxes (e.g. for 2d - x_min, x_max, y_min, y_max)
- Build complexity: *O(N log N)*
- Query complexity:
  Low-dim: *~O(log N)*
  High-dim: degenerates to *O(N)*
- Curse of dimensionality: poor pruning in high dimensions → almost linear scan



Tree-based methods

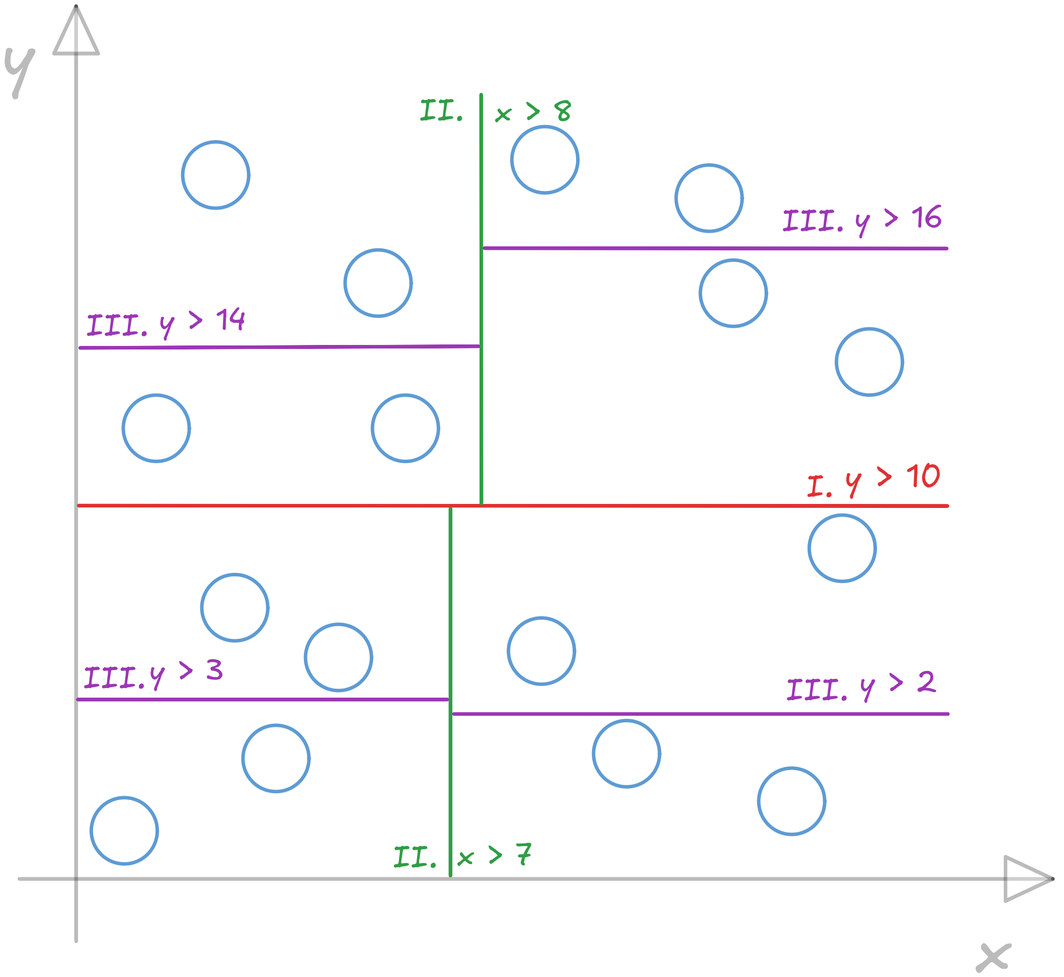
kd-tree

Build:

- Choose split dimensions (round-robin or max-variance)
- Split at the median to keep the tree balanced
- Store a bounding box for each node

Search:

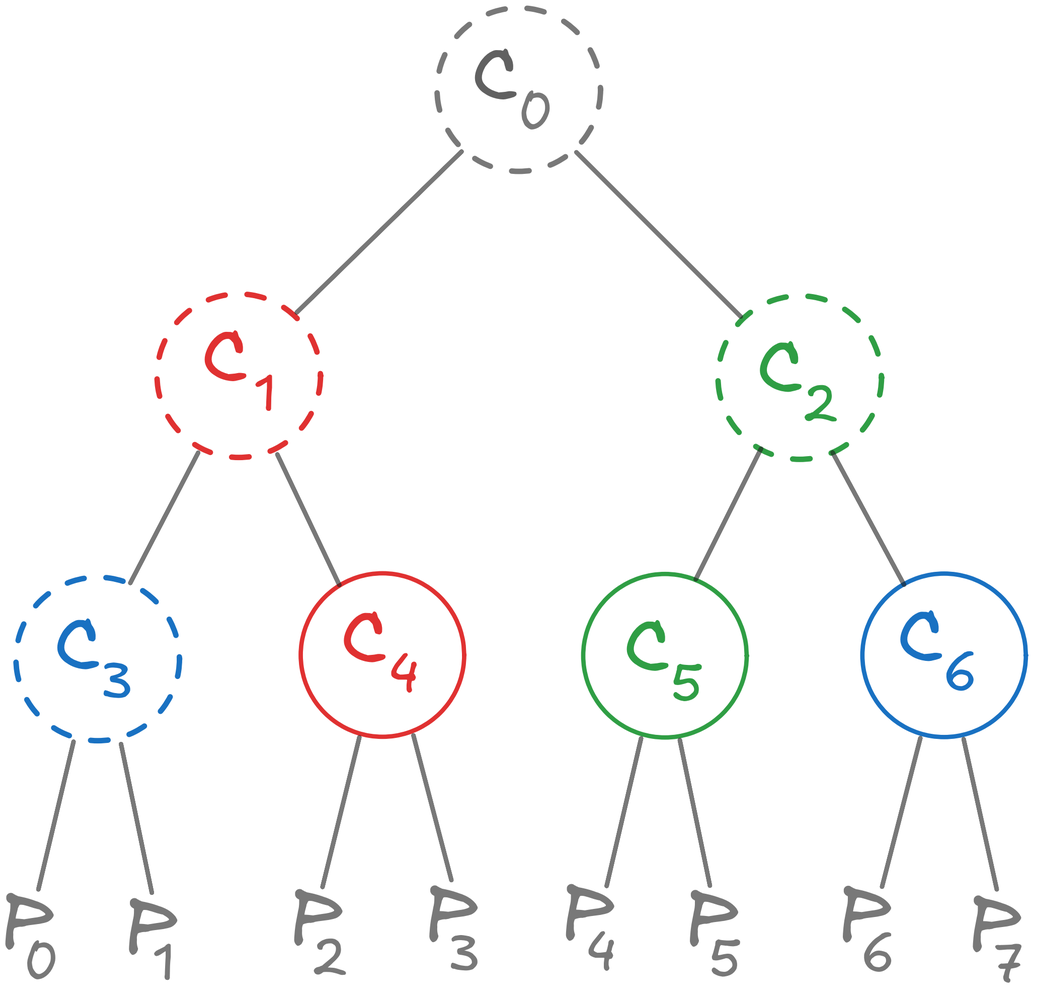
- Descend into the “near” child based on the query’s coordinate
- Evaluate leaf points to update the current best distance
- Backtrack into the “far” subtree only if its bounding box could contain a closer points
- Backtracking ends when no remaining subtree can possibly improve the result (bounding-box lower bound $\geq$ current best distance)



Tree-based methods

Ball tree

- Recursively splits points into clusters, each enclosed in a ball (center + radius)
- Each node stores a centroid and a radius covering all points
- Splits are chosen using distance-based (not axis-aligned)
- Build complexity: *O(N log N)*
- Query complexity:
  Low-dim: *~O(log N)*
  High-dim: degenerates to *O(N)*
- Curse of dimensionality: better than kd-tree, but pruning still fades with large dimensionality



Tree-based methods

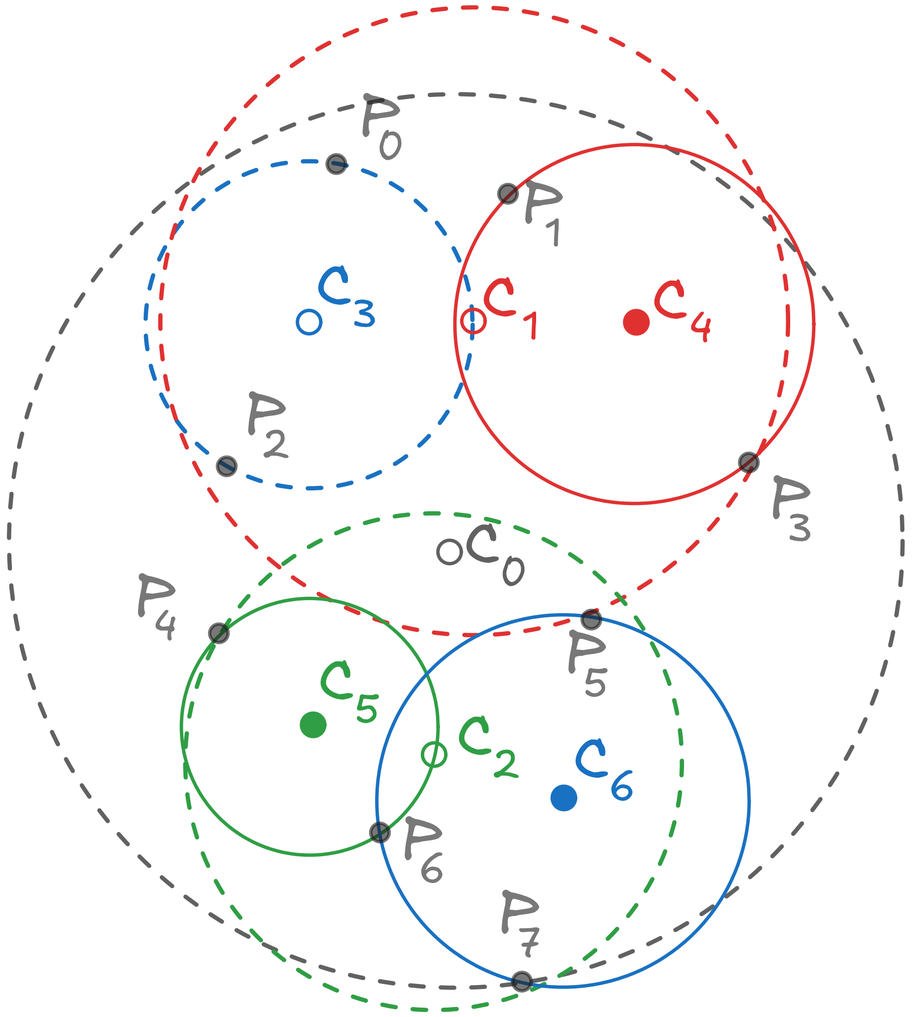
Ball tree

Build:

- Choose 2 “pivot” points (usually far apart)
- Assign all points to the closer pivot → 2 clusters
- For each cluster:
  - Compute centroid
  - Compute radius (max distance to centroid)
- Recurse until leaf size is reached

Search

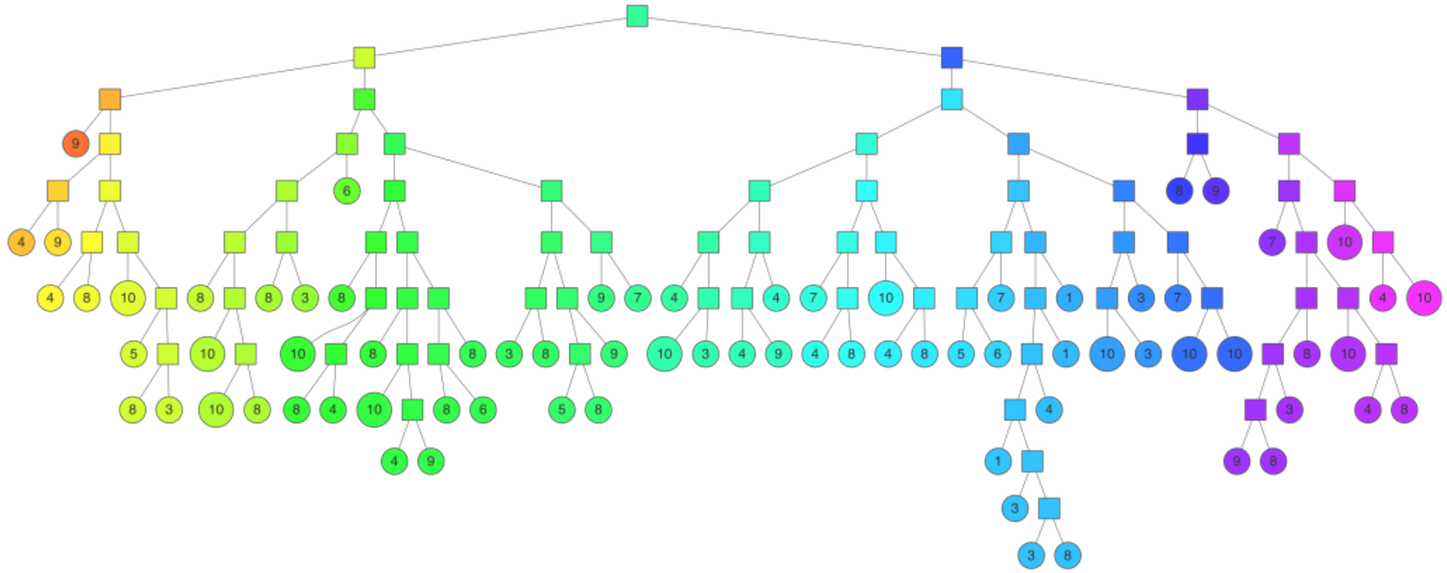
- Compute distance from query to node’s centroid
- Use triangle inequality ($d(q, c) - r \geq best\_dist$) to decide if the entire ball can be skipped
- Descend into nearer ball first
- Explore the other ball only if it could contain a closer point



Tree-based methods

Annoy

- Builds many randomised binary trees (typically 10-100+)
- Trees split using random hyperplanes (not axis-aligned or distance-based)
- Build complexity: $O(N \cdot T \cdot \log N)$, T - number of trees
- Query complexity: $O(T \cdot \log N)$
- Approximate by design; more trees → higher recall



Erik Bernhardsson: Nearest neighbours and vector models - part 2

spotify/annoy

Tree-based methods

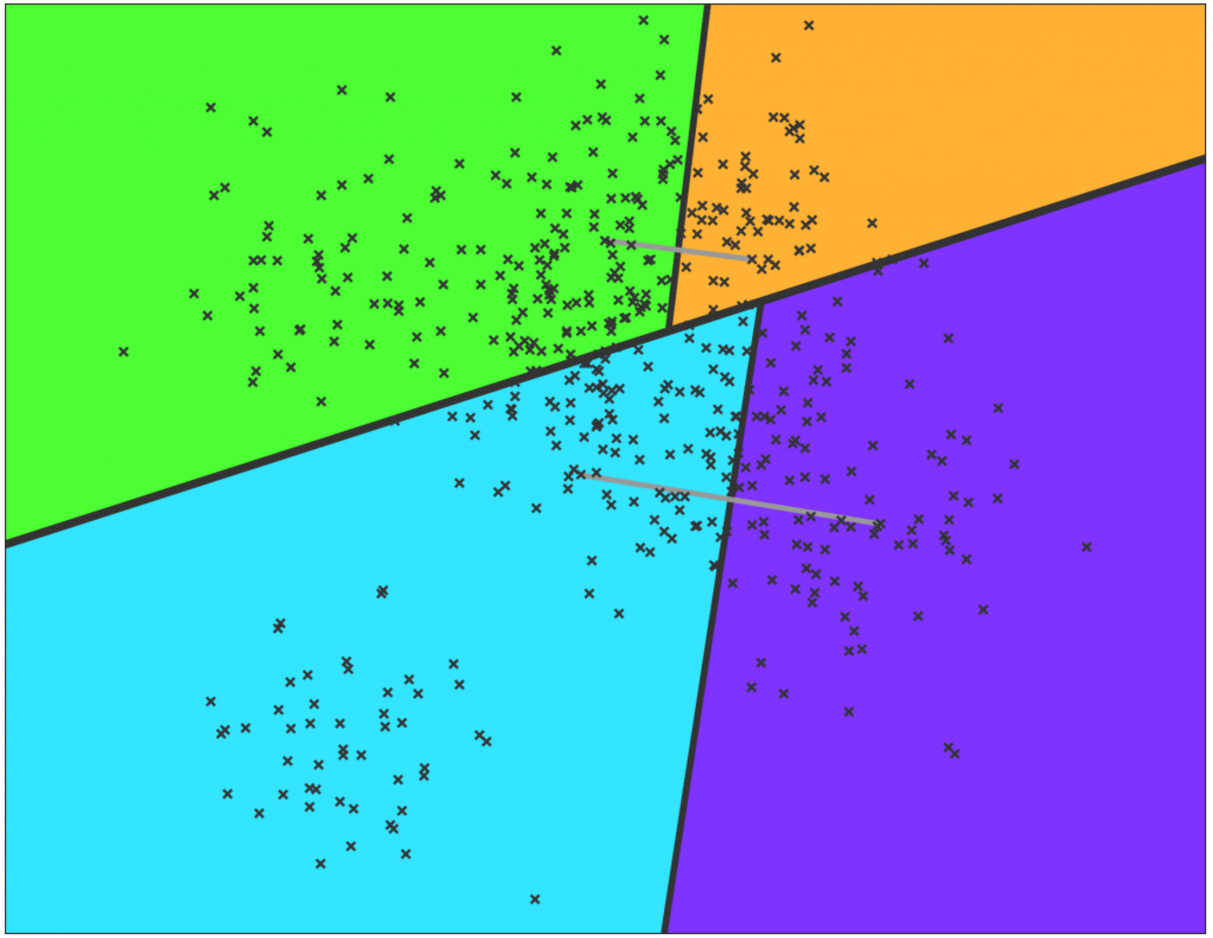
Annoy

Build:

- Select 2 random points far apart
- Split by the hyperplane equidistant from these points
- Recurse until leaf size is small
- Repeat to build many independent trees

Search:

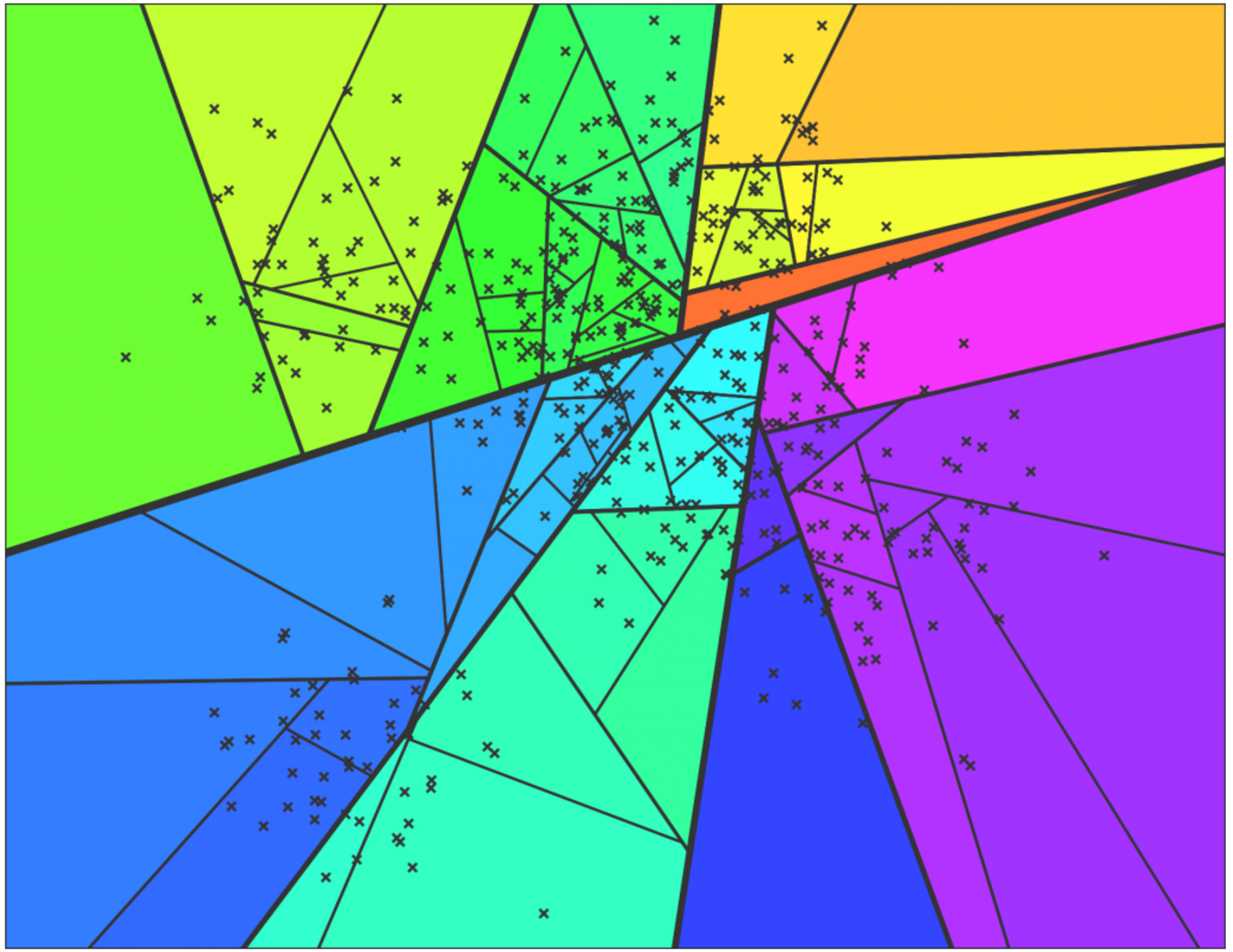
- For each tree:
  - Descend to the leaf using projection comparisons
  - Collect leaf points as candidates
- Merge candidates from all trees, compute exact distances



Tree-based methods

Annoy

- Builtin memmap support: loads trees directly from disk, allowing fast queries on large datasets with minimal RAM
- Used successfully on tens to hundreds of millions of vectors; disk size and build time scale with the number of trees
- Good for low-mid dim vectors (<1000), more trees needed as dim grows
- Once build, the index is read-only - no efficient inserts/deletes

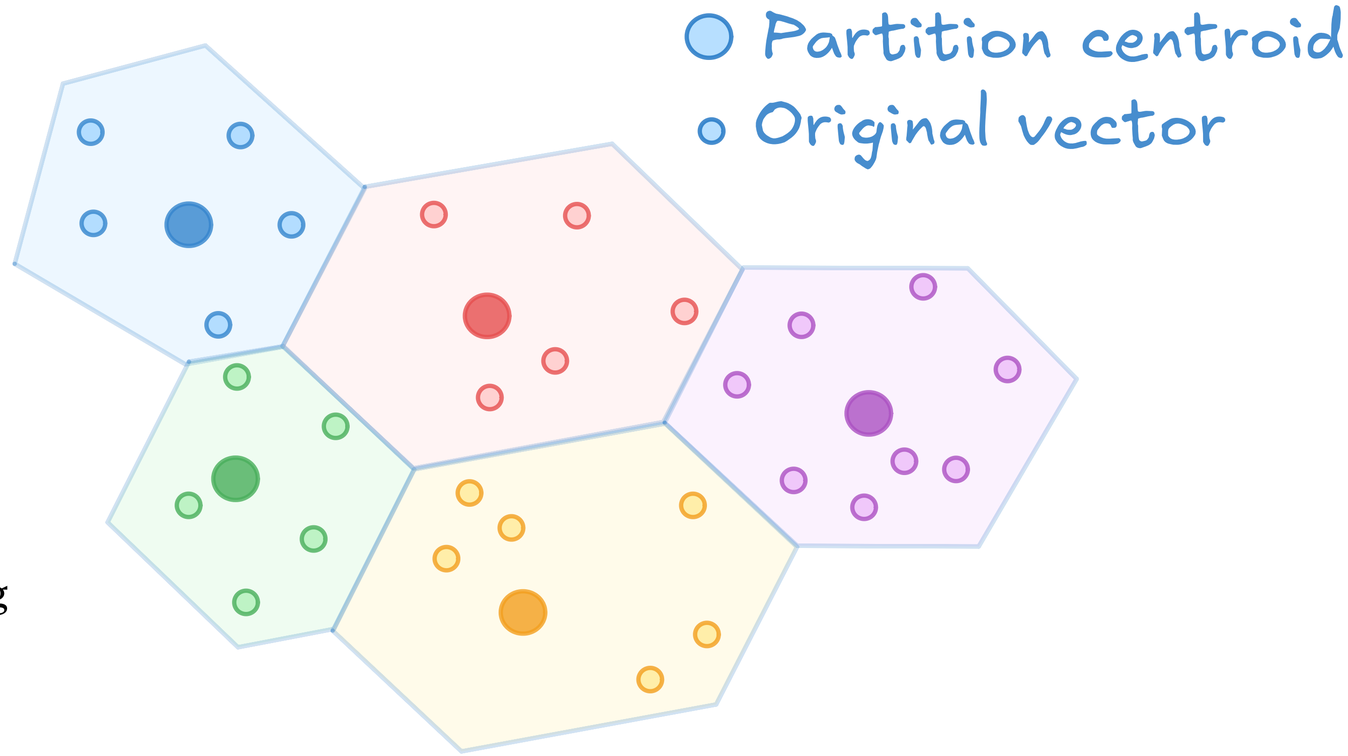


Erik Bernhardsson: Nearest neighbours and vector models - part 2

Clustering and quantisation methods (IVFPQ)

Inverted file index (IVF, IVFFlat)

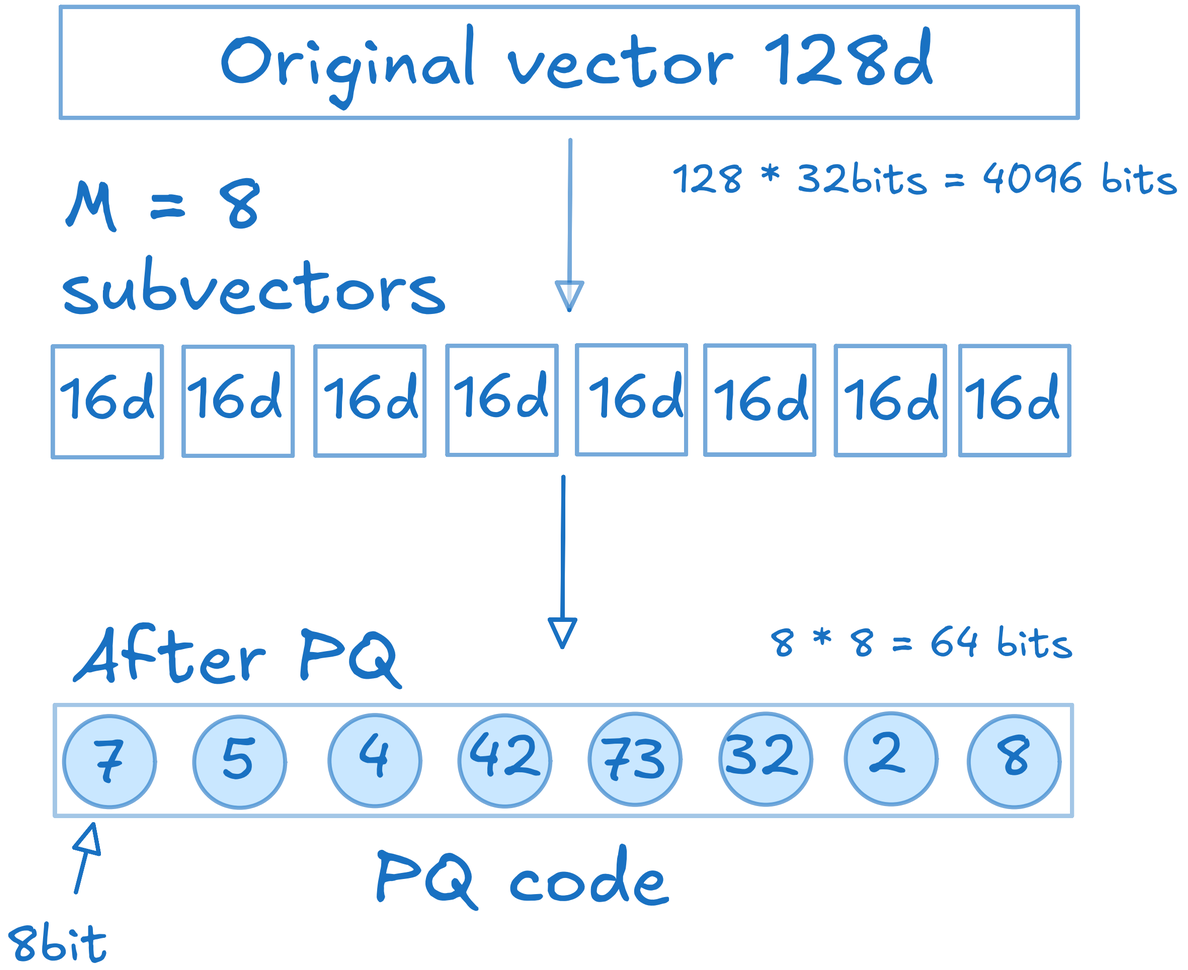
- Partition the vector space into k coarse clusters (via k-means)
- Assign each vector to its nearest centroid, forming inverted lists.
- At query time, search only the closest *nprobe* lists instead of the whole dataset.
- Reduces candidate from $N \rightarrow \frac{N}{k}$, giving sub-linear search
- Stores full vectors - improves speed, but not memory
- Accuracy/speed controlled by *k* (number of clusters) and *probe (lists to search)*
- Not ideal for frequent updates - inserting without re-training can degrade cluster quality
- Struggles with non-clustered distribution, depends on k-means quality



Clustering and quantisation methods (IVFPQ)

Product Quantization (PQ)

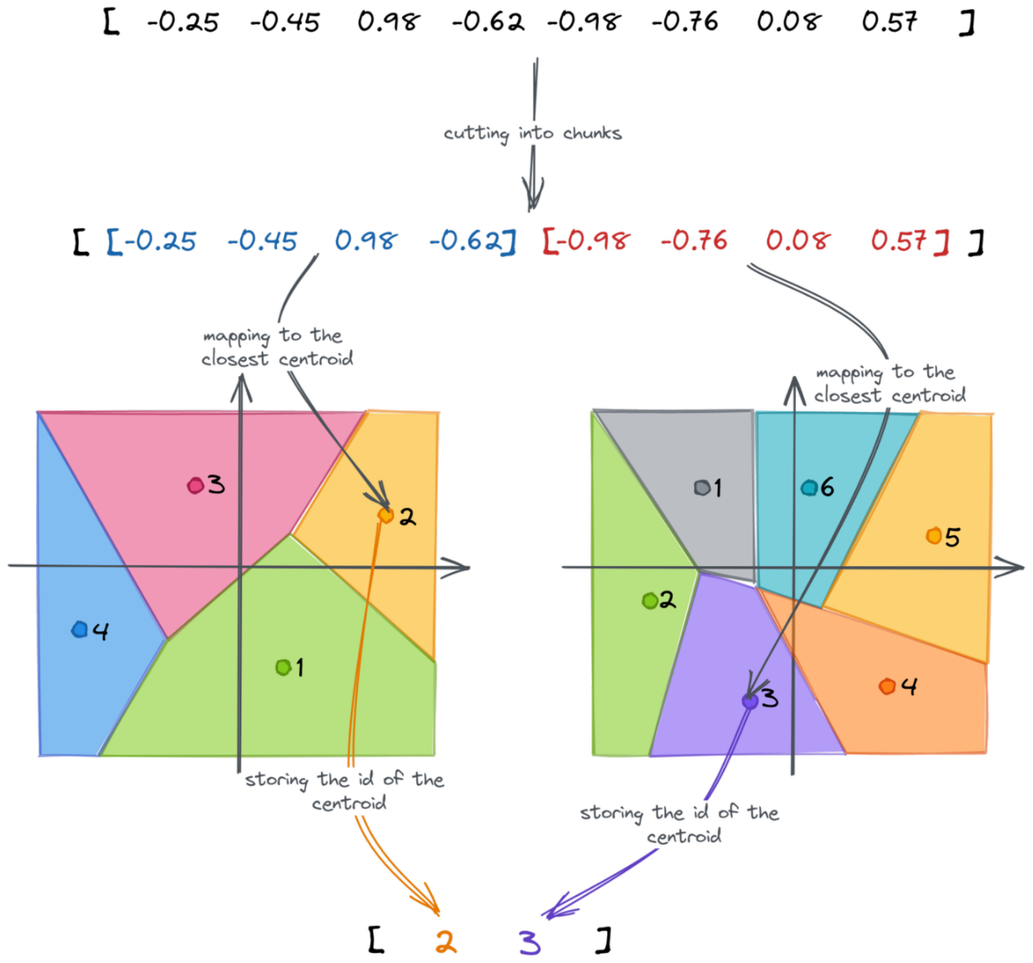
- Compress each vector into a few bytes by splitting it into *M* subspaces and quantising each chunk independently
- Fast approximate distance computation using lookup tables
- Huge memory savings (e.g. 3072b → 16 – 64b)
- Scales well to high-dimensional modern embeddings
- Lossy approximation significantly reduces recall (especially for small M)
- Insertions are problematic: new data must use existing codebooks → may cause drift; re-training requires rebuilding the whole index



Quantization based methods

PQ

- Split the vector into M equal sub-vectors (e.g., $D = 128 \rightarrow M = 8$, chunks of 16 dims)
- For each subspace train a small k-means with 2^n centroids if *nbits* = *n*
- Encode each chunk by storing the index of its nearest centroid → produces *M* 1-byte codes
- At query time, build lookup tables:
  For each chunk of the query, compute distances to all centroids
- Compute distance using ADC (Asymmetric Distance Computation)
  Total distance = sum of M lookup-table values
  Avoids reconstructing full vectors
- PQ reduces computation from *O(D)* to *O(M)* and memory from *D floats* to *M bytes*



Product quantization in vector search

Quantization based methods

IVFPQ

Idea:

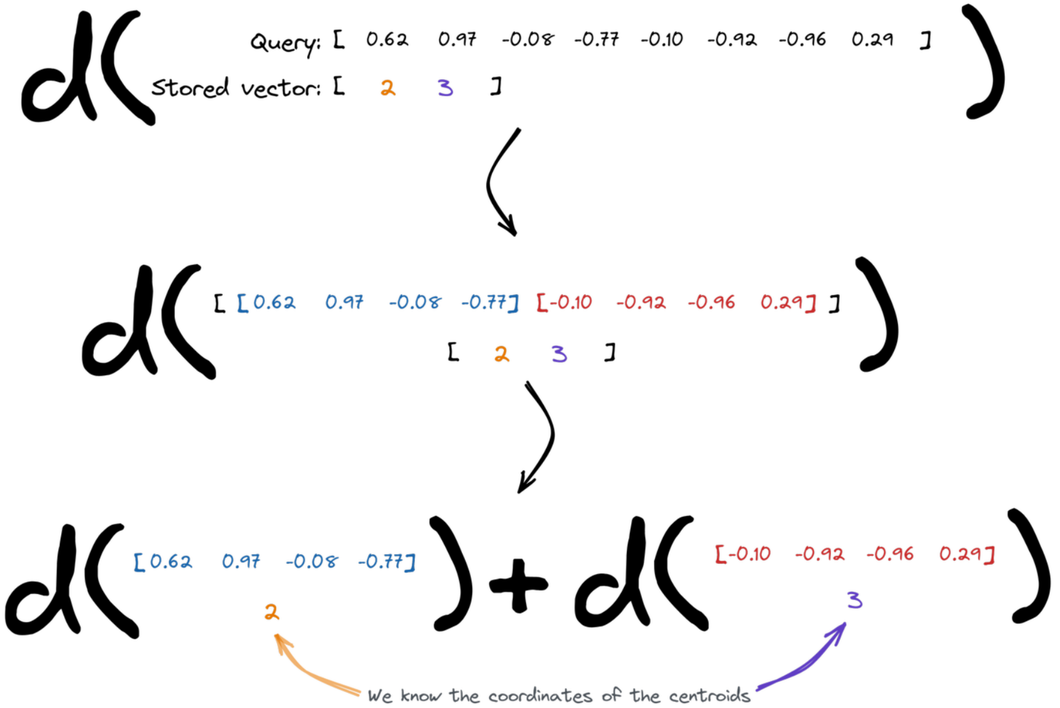
- Partition the dataset into coarse clusters (IVF)
- Store PQ-compressed vectors inside each cluster.

Build:

- Train coarse k-means (1st k-means) → inverted lists
- Train PQ codebooks globally (M small k-means) → centroids for each subspace
- Assign each vector to the nearest coarse centroid and PQ-encode it

Query phase:

- Find nearest coarse centroids (check *nprobe* cells)
- Search only those lists using PQ lookup tables (ADC)
- We often don't PQ the original vectors, but the residuals after subtracting the IVF centroid (easier to quantise)



Product quantization in vector search

Clustering and quantisation

IVFPQ Summary

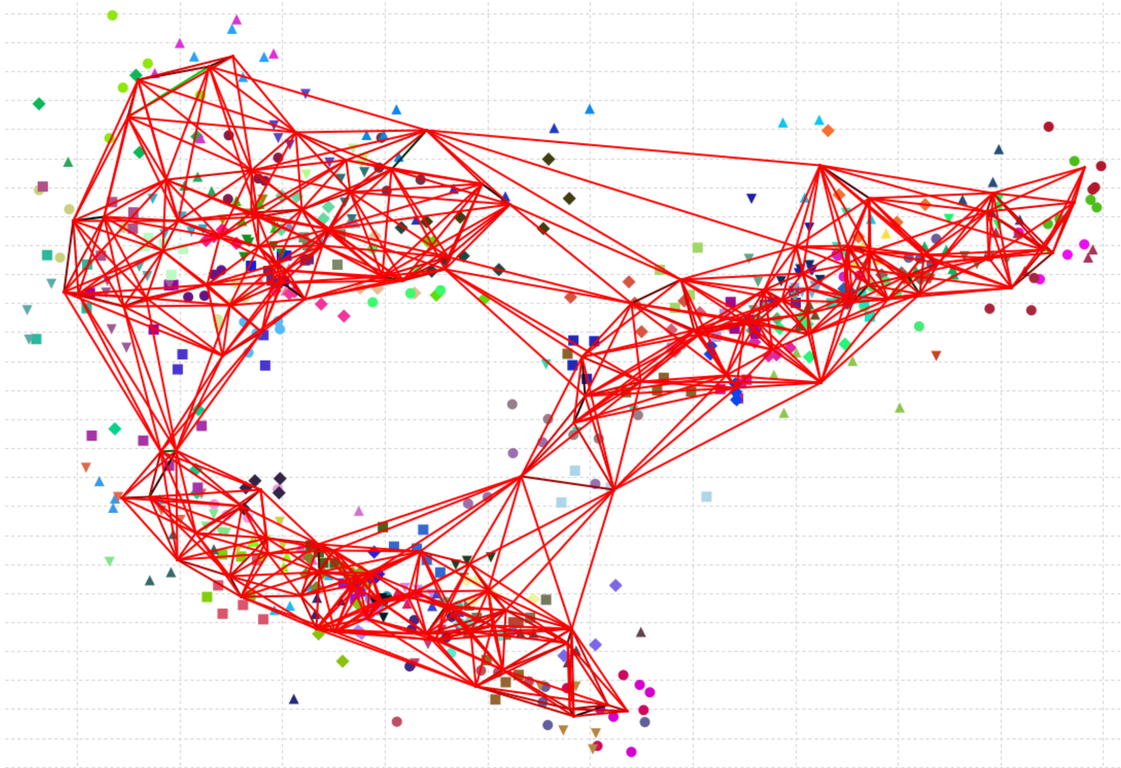
- Sub-linear search from IVF
- Memory savings from PQ
- Scales to hundreds of millions / billions vectors
- Fast distance evaluation via lookup tables (ADC)

- Quantization is lossy → recall is limited
- Insertions degrade quality unless retrained
- Performance depends on good clustering

Graph based methods

A different approach to search

- Represent vectors as nodes in a proximity graph
- Search by walking the graph, moving toward closer neighbours
- Greedy search explores only a small part of the graph
- Works well in high dimensions, unlike trees
- High memory usage (full vectors + graph links)
- Might be slow or expensive to build
- Examples: HNSW, NSG, Vamana (DiskANN)



Graph-based methods

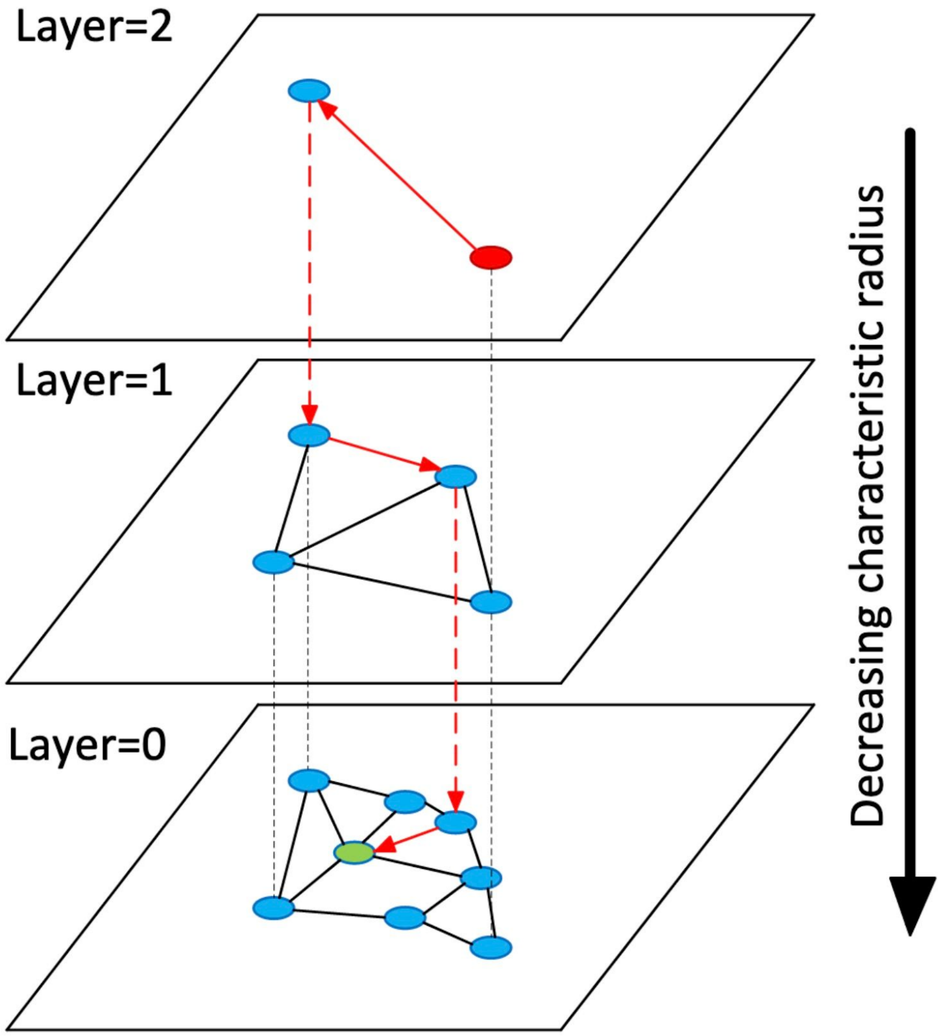
HNSW

Idea:

- Build a multi-layer proximity graph where higher layers provide long-range jumps, and lower layers provide fine-grained local neighbours.
- Search starts at the top and descends, always moving toward closer nodes

Key properties:

- Extremely fast, low-latency search
- Can achieve higher recall than the other non-graph based methods
- Works well in high-dimensional spaces



Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs

Graph-based methods

HNSW

Construction:

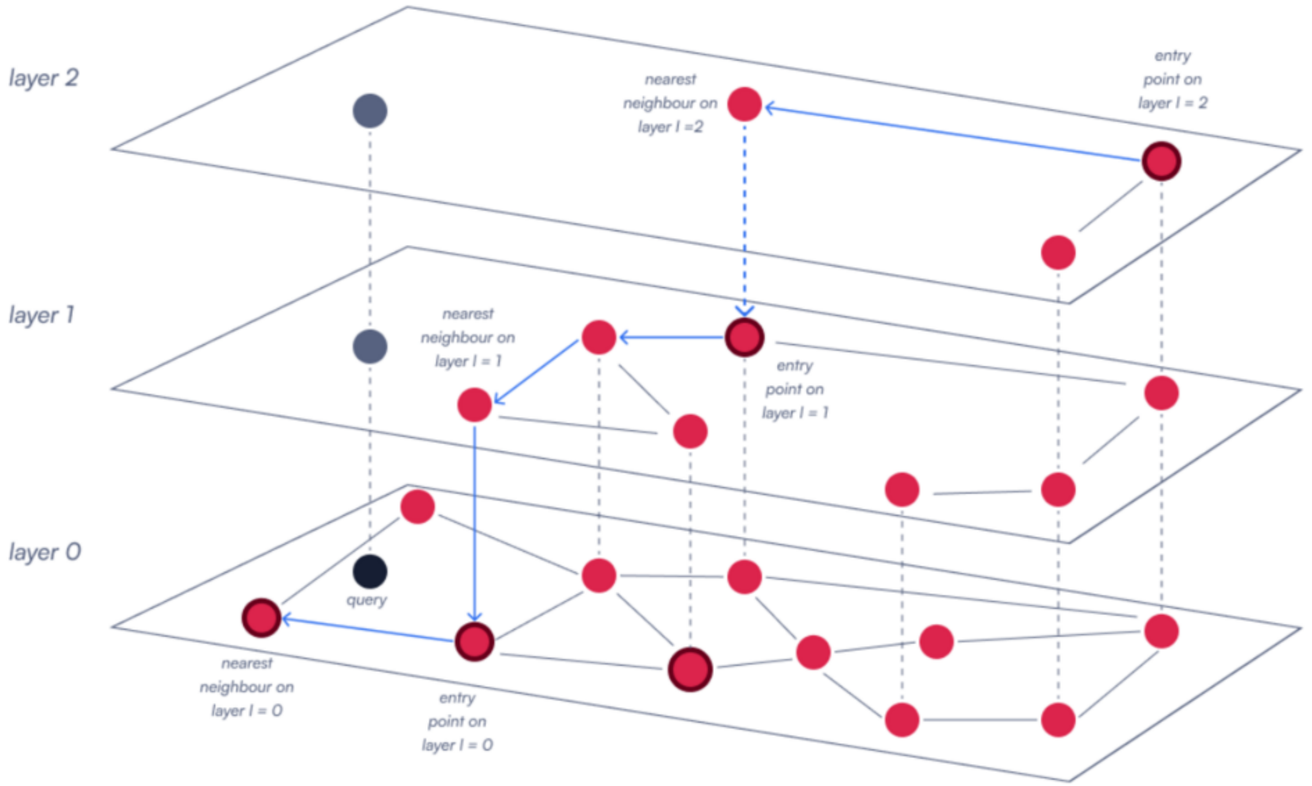
- Assign each point a random maximum layer (exponential distribution)
- Insert top-down: greedy search at each level until nearest entry point is found
- At level 0, connect to the closest M neighbours

Search:

- Start at a top layer
- Greedy descent until level 0

Key parameters:

- *m* - number of edges per node, the larger the value, the higher the precision, but more space required
- *efConstruction* - number of neighbours to consider during the index building, the larger the value, the higher the precision, but the longer the indexing time
- *efSearch* - number of neighbours to consider during the search, the larger the value, the higher the precision, but the longer the search time



HNSW indexing fundamentals

Graph-based methods

HNSW

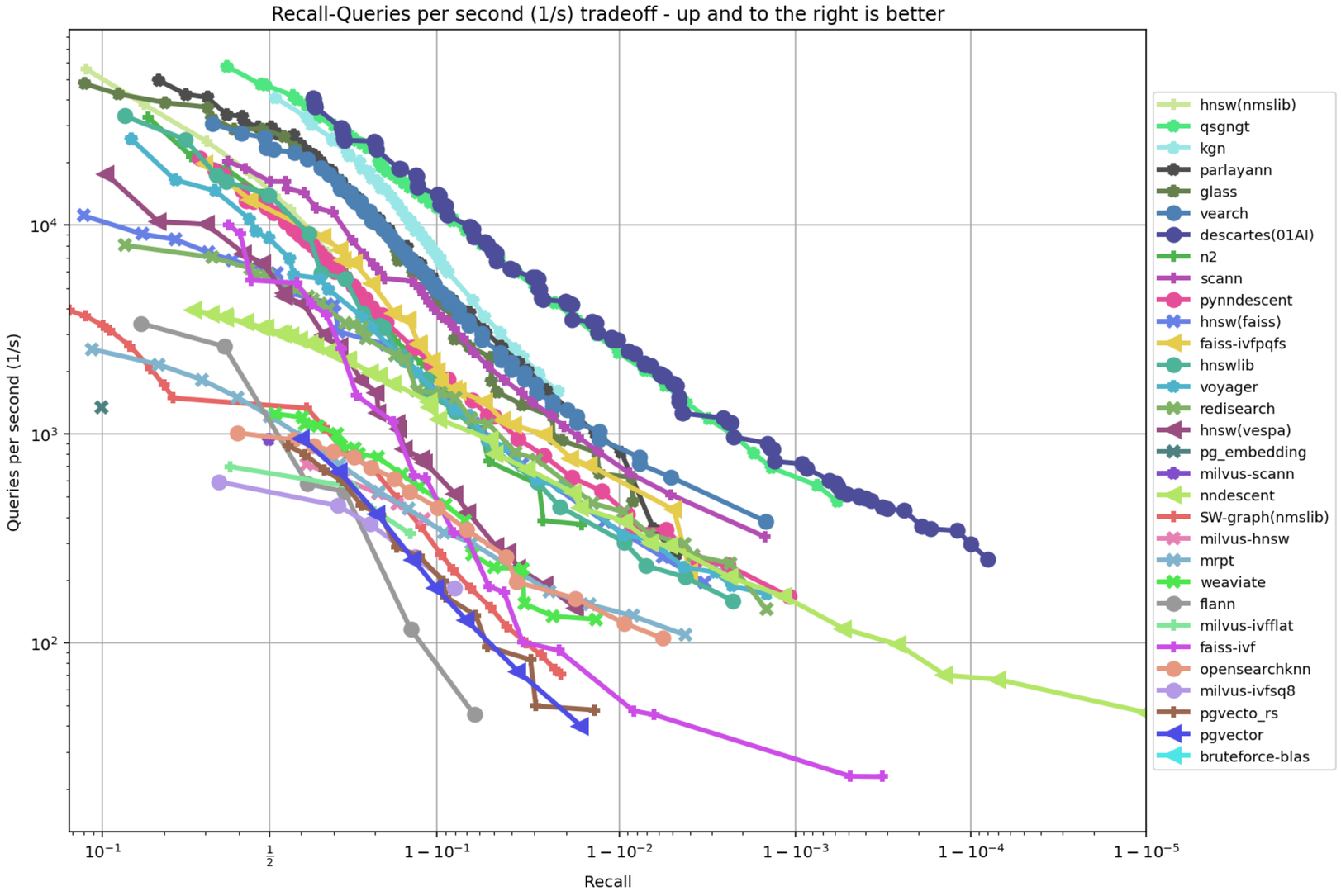
- High recall without quantisation
- Low-latency queries
- Handles incremental insertions
- Performs well on high-dimensional embeddings
- Graph construction might be done on a GPU

- High memory usage: full vectors + graph links for every node
- Build time might be long
- True removal requires rebuild

More in the next lecture

ANN Benchmarks

- Probably compares the biggest number of algorithms and vendors
- Update a couple of times a year
- All the competitors have the same restrictions (e.g. single-threaded environment, single thread products can look better than parallel-first ones)
- Usually low-medium dim vectors experiments



erikbern/ann-benchmarks

ANN benchmarks

Vendor provided

- Marketing affected, vendor that provides a benchmark will be on the first place most of the times
- Assumption: second best product across different benchmarks might actually be good
- Optimal setup for competitors might not be known by the developers of a benchmark

Engine	Setup	Dataset	Upload Time(m)	Upload + Index Time(m)	Latency(ms)	P95(ms)	P99(ms)	RPS	Precision
qdrant	qdrant-sq-rps-m-64-ef-512	dbpedia-openai-1M-1536-angular	3.51	24.43	3.54	4.95	8.62	**1238.0016**	0.99

Vector database benchmarks

Questions?