1. k-nearest neighbours

KNN (K-Nearest Neighbors) indexing is a brute-force technique used in the retrieval stage of RAG models.

Core Concepts:

- **Similarity Measure:** A function (e.g., cosine similarity, L2 distance) that determines the "closeness" between two vectors. The retrieval process aims to identify passages/documents whose vector representations are most similar to the query vector.

> *Note: Use L2 distance when the magnitude of features matters (Ideal when raw values and their differences are important, like comparing user profiles based on age and income) otherwise use cosine similarity (Ideal when comparing documents based on word topics, regardless of the overall word count).*

- **k Value:** The number of nearest neighbours to retrieve for each query. A higher k value provides more diverse results, while a lower k value prioritizes the most relevant ones.

Search Process:

1. **Query Vectorization:** The user's query is transformed into a vector using the same method applied to the corpus.
2. **KNN Search:** It retrieves the k nearest neighbour vectors from the corpus using given similarity metrics. Its time complexity is O(N log k) but using the KD tree for search will reduce the time complexity to O(log N)

2. Inverted File Vector

IVF indexing is an ANN technique used to accelerate the retrieval stage in RAG. It shares some similarities with KNN indexing but operates in a slightly different way.

Core Concepts:

- **Clustering:** The vector space is partitioned into clusters using techniques like k-means clustering. Each cluster has a centroid, which represents the center of that cluster in the vector space.
- **Inverted File:** An inverted file data structure (Just like a Python dictionary) is created. This file maps each centroid to a list of data point IDs (passages/documents) that belong to the corresponding cluster.

Search Process:

1. **Nearest Centroid Search:** The IVF index efficiently searches for the nearest centroid (the cluster centroid vector most similar to the query vector) based on a similarity measure (often cosine similarity).

Top highlight

2. **Refined Search:** Within the cluster identified by the nearest centroid, a smaller number of nearest neighbours (data points) are retrieved using a more expensive distance metric (like L2 distance). This step refines the search within the most promising cluster.

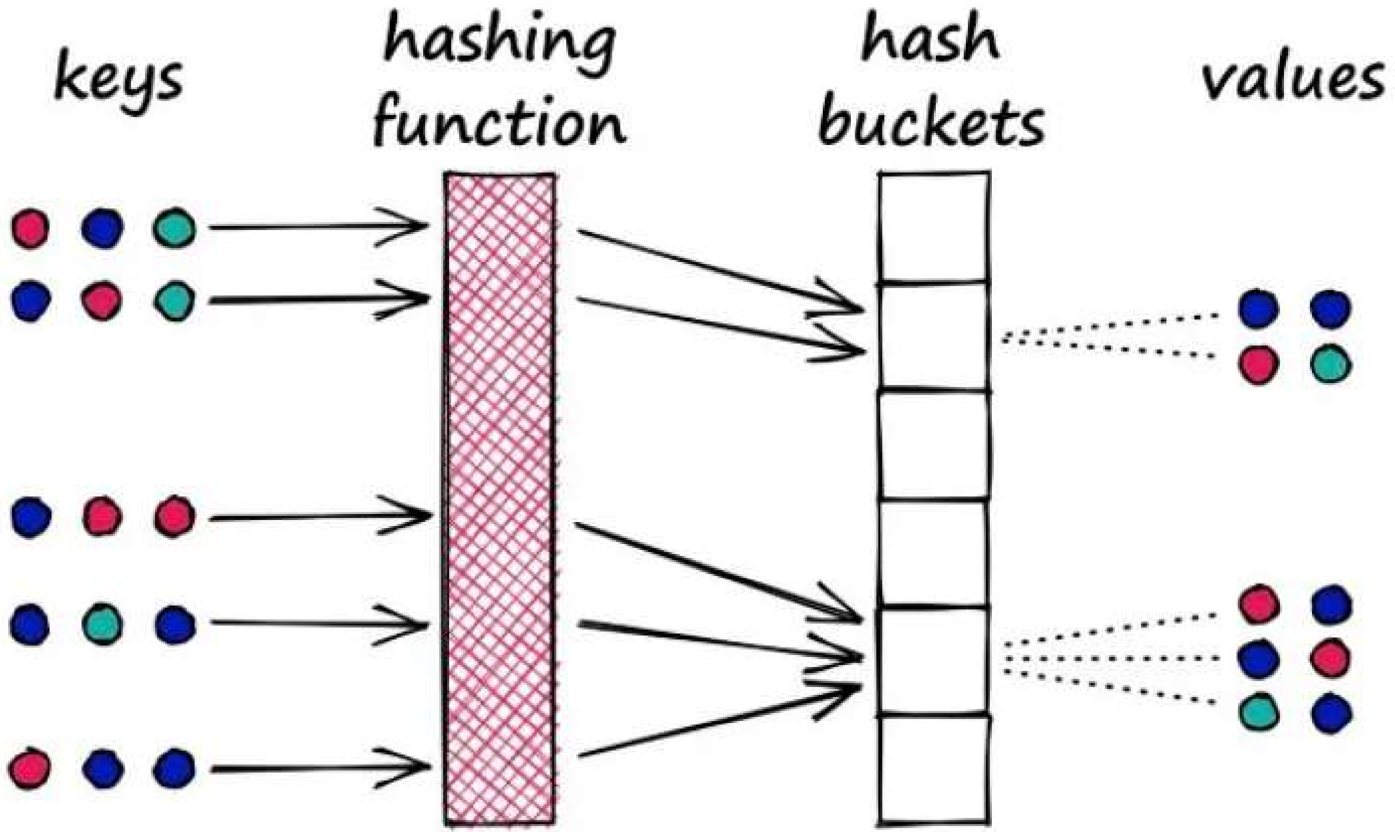
3. Locality Sensitive Hashing

LSH indexing serves to expedite the retrieval process. Unlike the previously discussed methods (KNN, IVF), LSH focuses on mapping similar data points to the same "buckets" with a high probability, even though it might not guarantee the closest neighbours.

Core Concepts:

- **Hash Functions:** LSH utilizes a family of hash functions that map similar data points (represented as vectors) to the same "hash bucket" with a high probability. However, dissimilar data points might also collide in the same bucket.

- **Similarity Measure:** A function (like cosine similarity) determines the "closeness" between two vectors. The LSH functions are designed to map similar vectors (based on the similarity measure) to the same bucket with a higher probability compared to dissimilar vectors.
- **Multiple Hash Tables:** LSH often uses multiple hash tables, each employing a different LSH function. This approach increases the likelihood of finding similar items even if they collide in one table, as they might be separated in another.



Search Process Steps:

1. **LSH Hashing:** The query vector is hashed using each hash function in the LSH family, resulting in a set of hash codes (bucket indices) for each table.
2. **Candidate Retrieval:** Based on the generated hash codes, documents that share at least one hash code (i.e., potentially similar based on the LSH function) with the query in any of the tables are considered candidate matches.

Goal of LSH

Find **similar items** quickly without comparing everything.

Example data (very simple)

Assume we have **numbers** instead of vectors:

```powershell
Data points: 12, 13, 25, 26, 40
```

Similar numbers are close to each other.

Step 1: Create multiple hash tables

Each table has a **different hash function**.

Hash Table 1

Hash function:

```cpp
h1(x) = x // 10
```

Hash Table 1

Hash function:

```cpp
h1(x) = x // 10
```

Buckets:

```
12 → bucket 1
13 → bucket 1
25 → bucket 2
26 → bucket 2
40 → bucket 4
```

Hash Table 2

Different hash function:

```cpp
h2(x) = (x + 3) // 10
```

Buckets:

```
12 → bucket 1
13 → bucket 1
25 → bucket 2
26 → bucket 2
40 → bucket 4
```

(Imagine in real cases these can differ more.)

Step 2: Query comes

Query = **14**

Hash the query using both functions:

```
h1(14) = 1
h2(14) = 1
```

So we check:

- Bucket 1 in Table 1
- Bucket 1 in Table 2

Step 3: Find candidates

From those buckets we get:

```
12, 13
```

These are **similar to 14**.

Why this works

- Similar values usually land in the **same bucket**
- Multiple tables give **multiple chances**
- We search only a **small part**, not all data

Very short summary

> LSH hashes data many times using different rules so similar items land in the same bucket in at least one table.

4. Random Projection

RP indexing projects high-dimensional data (text vectors) into a lower-dimensional space while attempting to preserve similarity relationships.

Core Concepts:

- **Dimensionality Reduction:** RP aims to reduce the dimensionality of textual data represented as vectors. High-dimensional vectors can be computationally expensive to compare.
- **Random Projection Matrix:** A random matrix is generated, with each element containing random values (often following a Gaussian distribution). The size of this matrix determines the target lower dimension.
- **Similarity Preservation:** The goal is to project data points (vectors) onto a lower-dimensional space in a way that retains their relative similarity as much as possible in the high-dimensional space (Binary form). Just like SVM, when the hyperplane normal vector produces a +ve dot-product with another vector, we encode it as 1 else 0. This allows for efficient search in the lower-dimensional space while still capturing relevant connections.

Search Process Steps:

1. **Random Projection:** Both the query vector and the document vectors in the corpus are projected onto the lower-dimensional space using the pre-computed random projection matrix. This results in lower-dimensional representations of the data (binary vector).
2. **Search in Lower Dimension:** Efficient search algorithms (Hamming distance to find the closest match) are used in the lower-dimensional space to find documents whose projected vectors are most similar to the projected query vector. This search can be faster due to the reduced dimensionality.

Problem

We want to find **documents similar to a query**.

Each document is a **vector**.

Step 0: Original data (high-dimensional)

Assume vectors have **4 dimensions** (small for understanding).

java

```java
Document D1 = [2, 1, 0, 1]
Document D2 = [2, 2, 0, 0]
Document D3 = [0, 0, 3, 3]

Query Q     = [2, 1, 0, 0]
```

Visually:

- D1 and D2 are similar to Q
- D3 is different

Step 1: Create a random projection matrix

We randomly generate a matrix to **reduce dimension from 4 → 2**

```java
Random Matrix R =
[ 1  -1 ]
[ 0   1 ]
[ 1   1 ]
[ -1  0 ]
```

(This matrix is fixed and used everywhere)

Step 2: Project all vectors (shrink them)

Multiply each vector by `R`.

Project Query

```java
Q × R = [2, 1, 0, 0] × R
      = [2, -1]
```

Project documents

java

Copy code

```java
D1 × R = [2, 1, 0, 1] → [1, -1]
D2 × R = [2, 2, 0, 0] → [2, 0]
D3 × R = [0, 0, 3, 3] → [0, 3]
```

Now everything is **2-D instead of 4-D**.

Step 3: Convert to binary (LSH step)

Use sign:

- positive → 1
- negative / zero → 0

css

Copy code

```css
Q   → [2, -1] → [1, 0]
D1  → [1, -1] → [1, 0]
D2  → [2,  0] → [1, 0]
D3  → [0,  3] → [0, 1]
```

Step 4: Fast search using Hamming distance

Compare **binary vectors.**

```ini
Q  = [1, 0]

D1 = [1, 0] → distance 0  ✅
D2 = [1, 0] → distance 0  ✅
D3 = [0, 1] → distance 2  ❌
```

Final result

Nearest documents to Q:

```
D1, D2
```

Correct answer found **without heavy computation** 🎯

5. Product Quantization

PQ indexing is used to accelerate the search process and reduce memory footprint.

Core Concepts:

- **Vector Decomposition:** High-dimensional query and document vectors are decomposed into smaller sub-vectors representing lower-dimensional subspaces. This effectively breaks down the complex vector into simpler pieces.
- **Codebook Creation:** For each subspace, a codebook is created using techniques like k-means clustering. This codebook contains a set of representative centroids, each representing a group of similar sub-vectors.
- **Encoding:** Each sub-vector is then “encoded” by identifying the closest centroid in its corresponding codebook. This encoding process assigns an index to the sub-vector based on its closest centroid.

Problem

We have vectors and want to **store + search fast**.

Assume vectors are **4-dimensional** (small for clarity).

Documents

```ini
D1 = [2, 4,  9, 10]
D2 = [3, 5,  8, 11]
D3 = [20, 18, 1,  2]
```

Step 1: Split the vector (PQ step)

We split each vector into **2 parts** (sub-vectors):

```sql
Part A → first 2 values
Part B → last 2 values
```

So:

```less
D1 → [2,4] | [9,10]
D2 → [3,5] | [8,11]
D3 → [20,18] | [1,2]
```

Step 2: Apply k-means (THIS is where k-means is used ✅)

We run **k-means separately on each part.**

For Part A (first 2 values)

Data points:

```css
[2,4], [3,5], [20,18]
```

Run k-means with **k = 2**

Clusters:

```css
Centroid 0 → [2.5, 4.5]
Centroid 1 → [20,18]
```

This becomes **Codebook A**.

For Part B (last 2 values)

Data points:

```css
[9,10], [8,11], [1,2]
```

Run k-means with **k = 2**

Clusters:

```css
Centroid 0 → [8.5, 10.5]
Centroid 1 → [1,2]
```

This becomes **Codebook B**.

Step 3: Encode each vector using centroids

Now we **replace values with centroid IDs.**

D1

```css
[2,4]   → closest to Centroid 0 (Part A)
[9,10]  → closest to Centroid 0 (Part B)

Encoded D1 → [0, 0]
```

D2

```css
[3,5]   → Centroid 0 (Part A)
[8,11]  → Centroid 0 (Part B)

Encoded D2 → [0, 0]
```

D3

```css
[20,18] → Centroid 1 (Part A)
[1,2]   → Centroid 1 (Part B)

Encoded D3 → [1, 1]
```

Step 4: Query example

Query:

```ini
Q = [3,4,  9,9]
```

Split:

```less
[3,4] | [9,9]
```

Encode:

```css
[3,4] → Centroid 0 (Part A)
[9,9] → Centroid 0 (Part B)

Encoded Q → [0, 0]
```

Step 5: Fast search

Compare encoded vectors:

```css
Q  → [0,0]
D1 → [0,0]  ✅
D2 → [0,0]  ✅
D3 → [1,1]  ❌
```

Closest documents:

```
D1, D2
```

Big idea (one line)

> **HNSW finds very similar items very fast, but not always the perfect best one — and that's okay.**

That's why it's great for **RAG, search, recommendations.**

Imagine this situation

You are in a **huge city** and want to find the **nearest restaurant**.

You don't check **every street** (too slow).
Instead, you:

1. First use **highways**
2. Then use **main roads**
3. Finally use **small streets**

That's HNSW.

Core idea 1: Graph (connections between points)

- Each data point (vector) is a **node**
- Nodes are connected to **similar nodes**
- Also, a few connections go to **far-away nodes**

Like a **social network**:

- Close friends (similar vectors)
- Some random friends (far but useful for jumping)

This helps you move quickly across the space.

Core idea 2: Layers (skip list idea)

HNSW has **multiple layers**:

Top layer

- Very few nodes
- Long-distance connections
- Used to **jump fast** to the right area

Middle layers

- More nodes
- Medium-distance connections

Bottom layer

- All nodes
- Very close connections
- Used for **fine matching**

How search works (step by step)

1️⃣ Start at the top layer

- Pick a random node
- Use long jumps to get **close to the query area**

(Like using highways)

2️⃣ Go down layer by layer

- At each layer:
  - Move to nodes that are **closer to the query**
  - Reduce the search area

(Like moving from main roads to streets)

3️⃣ Final selection

- At the bottom layer:
  - Pick the **closest few nodes**
  - These are your results

Get Plus ×

Example setup

Assume we have **points on a line** (easy to imagine):

makefile

Copy code

```makefile
Points: 1, 5, 9, 13, 18, 25
```

We want to find the point **closest to query = 14**.

Correct answer should be **13**.

Step 1: Build layers (idea only)

Top layer (few points, long jumps)

yaml

Copy code

```yaml
Top layer nodes: 1, 9, 18
Connections: 1 ↔ 9 ↔ 18
```

Bottom layer (all points, short jumps)

```
1 ↔ 5 ↔ 9 ↔ 13 ↔ 18 ↔ 25
```

Step 2: Start search at top layer

Start from a random node, say **1**.

Distances to query (14):

```
|14 - 1|  = 13
|14 - 9|  = 5   ← better
|14 - 18| = 4   ← better
```

Move:

```
1 → 9 → 18
```

Now 18 is the best in top layer.

Step 3: Go down to bottom layer

From **18**, move in the bottom layer.

Check neighbors:

```scss
18 → 13 (distance = 1) ← better
18 → 25 (distance = 11)
```

Move to **13**.

Step 4: Stop and return result

At 13, neighbors are worse:

```
|14 - 9|  = 5
|14 - 18| = 4
```

So we stop.

Result:

```mathematica
Nearest ≈ 13
```

What just happened (simple words)

1. Top layer helped **jump close fast**
2. Bottom layer helped **fine search**
3. We checked only a **few points**
4. We found a **very good answer quickly**

Why it's approximate

If connections were slightly different, it might return **18 instead of 13**.

But it would still be **very close**.

One-line summary

> HNSW first jumps close using top layers, then carefully searches nearby in lower layers.