Cairo University
Credit Hours system
Faculty of Engineering Computer Department

Phase 1 -Team 2

Mohamed Nabil Elsayed	1210291
Yusuf Ahmed Elsayed	1210330
Youssef Mohamed Abdelgawad	1210343

Submitted to: Eng / Muhammad Sayed
Date: 15/11 /2025

Semantic Search Engine

Introduction

The objective of this project is to design and implement an indexing system for a semantic search database that efficiently retrieves information based on vector space embeddings. The system will focus on building an indexing mechanism for a vector column.

Project Scope

The indexing system must:

- Store and index vectors (with 64-dimensional embeddings).
- Efficiently retrieve the top-k most similar vectors for a given query (with $k \leq 10$).
- Handle large datasets (up to 20 million vectors).
- Respond to queries within strict resource constraints.

Chosen Algorithm: IVF+PQ (Inverted File + Product Quantization)

We chose the **IVF+PQ** algorithm as it is one of the only established methods that can simultaneously meet the project's constraints on peak RAM usage and index size to retrieve results quickly without scanning the entire database.

How IVF+PQ Works:

This algorithm is a two-stage process:

1. **IVF (Inverted File):** First, the 20 million vectors are grouped into n clusters (e.g., 1000) using k-means. This acts like a filing cabinet, where each "folder" (inverted list) stores vectors that are already close to each other.
2. **PQ (Product Quantization):** This is a compression technique. It learns a "codebook" that shrinks each 64-dim vector into a tiny, approximate "code" (e.g., 10-20 bytes). This compression is what allows the entire 20M vector index to fit within the 200MB disk limit.

When searching, we find the query's closest "folder," load *only that single, compressed folder* into memory (respecting the 50MB RAM limit), and quickly scan its compressed codes to find the best approximate matches.

Architecture:

1. **Index Building:**
    a. **Centroid Training:** K-means clustering algorithm is used to train on a sample of the data to find the cluster centers (e.g., 1000 centroids).
    b. **PQ Codebook Training:** We train a compressor by "chopping" vectors into sub-vectors and running k-means on each to create a codebook.
    c. **Data Partitioning & Compression:** We iterate through the full database, assign each vector to its nearest centroid, compress it using the PQ codebook, and append its (compressed_code, row_id) to the correct cluster's file on disk.

2. **Retrieval:**
    a. **Load Headers:** Read the small centroids and pq_codebook files from disk into memory.
    b. **Cluster Probing:** Compare the query vector to the in-memory centroids to find the closest

cluster(s).

c. **Disk-Read:** Load *only* the compressed inverted list file for that specific cluster from disk. This is the key to obeying the "no caching" and 50MB RAM rules.

d. **Approximate Search:** Quickly compare the query vector to the compressed codes in memory and find the top k candidates.

e. **Re-ranking:** Fetch the *full, original* vectors for these 10 candidates. We then calculate the *exact* cosine similarity and return the true top-k.

Key Components:

1. **Index Class:** Manages the index building and retrieval logic.
2. **Centroids (on-disk):** A NumPy array (n_clusters x 64) storing the IVF cluster centers.
3. **PQ_Codebook (on-disk):** A NumPy array storing the compression data.
4. **Inverted_Lists (on-disk):** A set of binary files, one for each cluster, storing the (compressed_code, row_id) tuples.

Data Structure:

The index is composed of several files stored on disk:

1. **Centroids:** A NumPy array of shape (n_clusters, 64). This is loaded into memory during retrieval and is within the 50MB RAM limit.
2. **PQ Codebook:** A NumPy array storing the compression codebook. This is also loaded into memory and is very small.
3. **Inverted Lists:** A directory of n_clusters binary files. Each file contains a list of the compressed vectors and their original row_ids. During retrieval, only one f these small files is read from disk at a time, never loading the whole index into memory, thus satisfying the "no caching" rule.

Algorithm:

1. Build Index:

1. Sample the database and train a Mini-Batch K-means with n clusters.
2. Train the PQ codebook on a sample of the data.
3. Save the centroids and pq_codebook to disk.
4. Create n clusters empty files (inverted lists).
5. Iterate through the *full* database:
    a. For each vector, find its nearest centroid.
    b. Compress the vector using the PQ codebook.
    c. Append the (compressed_code, row_id) to the inverted list file.

2. Retrieve Top-K:

1. Load centroids.npy and pq_codebook.npy from disk into NumPy arrays.
2. Compare the query_vector to the centroids to find the nearest cluster.
3. Load *only* the corresponding files from disk.
4. Calculate *approximate* distances between the query and all compressed codes from that file.
5. Select the top k_candidates row_ids.
6. For each row_id in k_candidates:
    a. Fetch the *full original vector* from disk.
    b. Compute the *exact* cosine similarity.
7. Sort the exact scores and return the IDs of the top-k vectors.

Trade-offs:

1. **Speed vs. Accuracy:** This is an **A**pproximate **N**earest **N**eighbor (ANN) search. We trade perfect accuracy for massive speed, memory, and disk savings. Accuracy is "lost" at the clustering step (a vector's true neighbor might be in another cluster) and at the compression step.
2. **Index Size vs. Accuracy:** The main trade-off. A more aggressive PQ compression (e.g., 8 bytes per vector) creates a smaller index that fits the 200MB limit, but it is less accurate. A less aggressive compression (e.g., 16 bytes) is more accurate but may be too large.
3. **Build Time vs. Query Time:** Using more clusters (e.g., 4096 vs 1024) makes the `build_index` step take longer but makes `retrieve` *faster* because the inverted lists are smaller.
4. **Static Index:** Adding new vectors requires a full or partial index rebuild.
5. **Disk I/O vs. RAM:** This is our primary design choice. We explicitly choose to read from disk during the `retrieve` call to stay under the 50MB RAM limit, as per the project constraints.
6. **Implementation Complexity:** This algorithm is complex to implement from scratch, as it requires building both the IVF partitioning and the PQ compression logic using numpy and `scikit-learn`.