Cairo University
Faculty of Engineering
CMP 4030

Advanced Database

Semantic Search Engine

Phase #2

Team Members:

Name	Sec	BN	Code
Ahmed Hany Farouk	1	10	9202213
Mohab Zaghloul	2	28	9203568
Abdelrahman Fathi	2	2	9202846
Nour Ziad Almulhem	2	30	9204005

Presented to: Eng. Abdelrahman Kaseb

After investigating many search algorithms, we would like to provide what we have implemented so far.

1. Our approach

We used the inverted file for indexing with number of clusters differs with database size.

100 for 10k databases → around 100 vector per cluster

1000 for 100k databases → around 100 vector per cluster

1000 for 1m databases → around 1000 vector per cluster

5000 for 5m databases → around 1000 vector per cluster

5000 for 10m databases → around 2000 vector per cluster

10k for 15m databases → around 1500 vector per cluster

10k for 20m databases → around 2000 vector per cluster

For large database sizes, we used partial training. (Trained using part of the data)

Example:

- For database sizes less than or equal to 100k vectors, we used the whole dataset to get our centroids and cluster names.
- For databases larger than 100k vectors, we used random 100k vectors as the training dataset to get our centroids. Then predict the remaining data.

We searched in those clusters using cosine similarity following this technique:

We searched for 30 nearest centroids, then searched in each cluster of those for the 30 nearest vectors, then concatenated the search space and got the top-k results.

Note: At each cosine similarity step, the maximum number of vectors loaded into the memory is 2000 vectors.

- **Clustering**
  Start by dividing the database into clusters. The number of clusters varies depending on the size of the database. Each cluster is represented by a centroid, which is a point in the data space. Using a dataset (either the whole dataset for databases of size 100k or less, or a random sample of 100k vectors for larger databases) to train a clustering algorithm and find the centroids.
- **Inverted Indexing**
  For each data point, calculate which cluster it belongs to (i.e., which centroid it is closest to) and add it to an inverted index under that cluster's name. The inverted index allows you to quickly find all data points that belong to a certain cluster.
- **Searching**
  When querying a data point, first find the 30 nearest centroids. For each of these centroids, you look up the corresponding cluster in the inverted index and find the 30 nearest vectors within that cluster. Finally, concatenate these vectors and find the top-k nearest vectors to the query.

This approach is efficient because it reduces the number of distance calculations you need to perform. Instead of comparing the query to every data point in the database, you only compare it to a small number of centroids and a subset of the data points within the nearest clusters. However, as with any nearest neighbor search algorithm, there is a trade-off between speed and accuracy. The accuracy of the results depends on how well the clustering algorithm can group similar data points together.

Other Approaches we tried:

1. **LSH**

   1. Hashing: Start by creating 16 random vectors. These vectors are used as hash functions. Each vector represents a hyperplane in the space of your data.

   2. Cosine Similarity: For each data point, calculate the cosine similarity with each of the 16 random vectors. The cosine similarity measures the cosine of the angle between two vectors. This value will be close to 1 for similar vectors and close to -1 for dissimilar vectors.

   3. Thresholding: Apply a threshold to the cosine similarity to convert it into a binary value. If the cosine similarity is greater than the threshold, you assign a value of 1, otherwise, you assign a value of 0. This step effectively determines which side of the hyperplane (represented by the random vector) the data point falls on.

   4. Bucketing: Take the 16 binary values obtained for each data point and treat them as a binary number. This binary number is used as the index of the bucket to which the data point is assigned. All data points that end up in the same bucket are similar.

   This approach can significantly speed up nearest neighbor search because you only need to compare a data point with other points in the same bucket, rather than with all points in the dataset. However, the trade-off is that it is a probabilistic method, and it may not always give the most accurate results. The accuracy depends on the choice of the random vectors and the threshold.

2. **LSH forest**

   We tried this approach to make more than an LSH Table, but the accuracy was worse than the IV approach.

   The idea is to make what you did to LSH but more than one time in different tables and different hash vectors.

   1. Multiple Hash Tables: Instead of creating a single LSH table, you create multiple LSH tables. Each table is built with a distinct set of random vectors. This increases the chance that similar items will end up in the same bucket in at least one of the tables.

2. Hashing, Cosine Similarity, Thresholding, and Bucketing: These steps are the same as in the basic LSH algorithm, but they are performed separately for each hash table. Each data point ends up in one bucket in each table.

3. Querying: When querying a data point, you look in the corresponding bucket in each table. The union of the items in these buckets is considered as the set of items like the query.

The idea behind this approach is that by using multiple hash tables, you increase the probability of finding the true nearest neighbors of a query. However, this comes at the cost of increased storage and computation, as you need to maintain and probe multiple hash tables.

3. HAN

The idea in this approach is like LSH but instead of using a table using a KDTrees. which means a tree for each hash value. and the rest is like the LSH approach.

It gave a great timing but compared to IV approach, IV approach had better accuracy.

1. Hashing, Cosine Similarity, Thresholding: These steps are the same as in the basic LSH algorithm. You create random vectors, calculate the cosine similarity of each data point with these vectors, and apply a threshold to convert the cosine similarity into a binary value.

2. KD-Tree Construction: Instead of assigning each data point to a bucket based on its binary value, you insert it into a KD-Tree. A KD-Tree is a binary tree where each node is a k-dimensional point. All points to the left of a node are less than the node in one dimension, and all points to the right are greater. You create a separate KD-Tree for each unique binary value (hash value).

3. Querying: When you query a data point, you calculate its binary value (hash value), find the corresponding KD-Tree, and perform a nearest neighbor search in that tree. This is much faster than a brute force search in high-dimensional spaces.

The advantage of this approach is that it combines the dimensionality reduction of LSH with the efficient nearest neighbor search of KD-Trees. However, as with the other methods, there is a trade-off between speed and accuracy. The KD-Tree can only guarantee finding the nearest neighbor within its own set of points (those with the same hash value), so if the nearest actual neighbor has a different hash value, it will not be found. This could explain why the IV approach still has better accuracy.

Product Quantization

In this approach, we depend on the open-source library **nanopq.**

The product quantization depends on dividing the original space of vectors into M subspaces. Applying K-means on each subspace to quantize each sub vector into a single integer. Hence representing the (N, D) matrix using a (M, D) matrix called PQ-codes.

Our algorithm has two stages:
- Training stage
- Searching stage.

Training

- Dividing the original space into one million sections. Each section has twenty vectors.
- Each section is represented by a single vector (By averaging twenty vectors).
- Now we have one million records. This is the training data.
- Train and encode the one million records.
- Dump the codewords into a pickle file (70 MB).

Search

- Pick a random record from the database to be our query vector.
- Remove the id from the vector and save it for later check in the end.
- Load the pq_codes from the pickle file.
- Compute the distance table given the query vector (256, 70).
- Compute the distance vector given the pq_codes.
- Append distance vector to the IDs vector.
- Sort the matrix and pick the top ten.
- Use the IDs to get the original vectors of each section (two hundred vectors at max).
- Compare the query vector with each of the vectors.
- Select the nearest vector.