Cairo University
Faculty of Engineering

Department of Computer
Engineering

Advanced Database Project

Semantic Search Engine

Team 10

Name	ID
Eslam Ashraf	9202256
Beshoy Morad Atya	9202405
Abdelrahman Hamdy	9202833
Abdelrahman Noaman	9202851

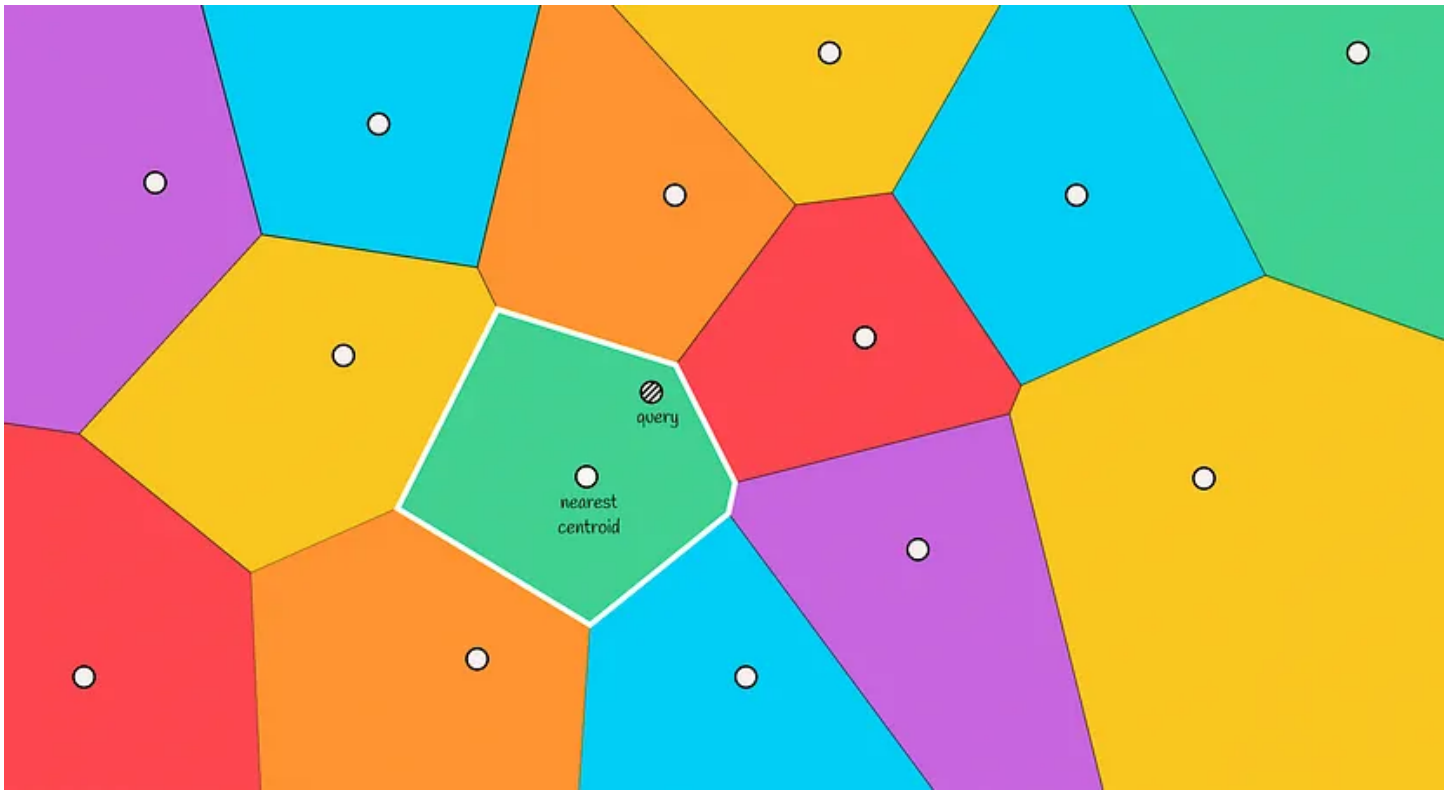
Date: 11/15/2023

Presented to: Eng. Abdelrahman Kaseb

Inverted File Index (IVF)

Approach:

Given a numpy array with a specific data size, we write the vectors in a csv file without their ID. The index in the csv file for each vector identifies its ID when searching. For data sizes of 1 million or less, we extract centroids from all vectors and its corresponding labels using the kmeans2 function from scipy library. Each vector label represents the centroid number it's assigned to.



We set the number of clusters and probing count according to the data size. So for example, 1 million vectors are split into 200 clusters, where each cluster has some vectors grouped together to be used for retrieval later on. Given a query, we search for the nearest centroid (located in the green zone).
And since we apply probing, we also look for neighboring clusters in case there are similar vectors on the edge of some.
By computing the similarity between each vector in the cluster and query, the final answer is returned.

- ***How do we build the index?***

First of all, we initialize 2 maps representing the inverted index, which contains lists of vectors grouped together according to their cluster number as index in this map. The second one is for the centroid vectors itself. Along with each centroid vector, we save the count of the number of vectors it contains and a previous count which means the number of vectors above that cluster in this map. So one entry looks like this: [Centroid vector, count, previous count]

- ***Why is this useful?***

In retrieval, when we've located a centroid to be nearest to the query, we only load the vectors inside it from the index file. To do that, we need a pointer at the beginning of this file and move that pointer to the exact location where the vectors start for this particular cluster.
Here, we benefit from the numpy memory map structure which basically helps to avoid loading the whole index file when searching and flooding the ram. It helps in implementing the pointer seeking functionality I've just explained.
In the index file, we have a node class which represents each element in a list. This node contains the complete vector data [1x70] and its id. So that when we retrieve, we can easily obtain the id by just indexing it. This whole process saves up time and ensures a fast retrieval of similar vectors, whatever the query is.
Files are stored in a binary format so that we read and write to them quickly and efficiently. I've tried working with csv files for the index, but they consume too much space and are slow in I/O operations.
Now the main question is.. How do we deal with data sizes 5m, 10m, 15m and 20m? Surely we don't store all these vectors in the ram right?
Well, big data sizes are treated differently.
We first extract centroids from only a portion of this huge data for example, first million vectors of the 5m data. And this is because the kmeans function explodes when large data is input to it.
This means that, we only have labels for the first million of each big data size only.. That's right.
For the rest of the data, we compute cosine similarities manually and assign each vector to one of the extracted centroids from the previous step. This is done by performing vectorized operations on each chunk to easily obtain the remaining labels (without loops).
Now what's left is to store these assignments in the index map, and increment the counts and previous counts of the corresponding centroids.
The huge data sizes are processed in chunks, so that when computing dot products in the similarities, the ram doesn't get full.
For each chunk, we update the index map and save it to the index file after processing all chunks.
The time for searching in 20m is **~0.8.**
The probing count varies according to the data size as well.

Size	Number of Clusters	Consumed RAM	Time (in seconds)	Recall (Diff seeds)
10k	**16**	**2.4 MB**	**0.04**	**0**
100k	**64**	**4.95 MB**	**0.05**	**0**
1M	**200**	**20 MB**	**0.2**	**0**

5M	750	60 MB	0.5	0
10M	1024	50 MB	1.04	0
15M	1536	98 MB	1.22	0
20M	2048	140 MB	1.74	0

In retrieval: We load the centroids and their counts from the centroids file.
Compute similarities between the centroid vectors and query, using numpy and vectorized operations.
Get the nearest n_probe centroid indices to this query.
Loop over these indices. For each one, get the vectors of the current centroid using the memory map, start index = previous count of vectors and the end is the start + count inside the cluster.
Again compute similarities and get the top k similar vectors to the query from all the probing.
Extend the result in a list and select the ids to be returned as the final result.

IVF is our main indexing algorithm and what we went for in the end.

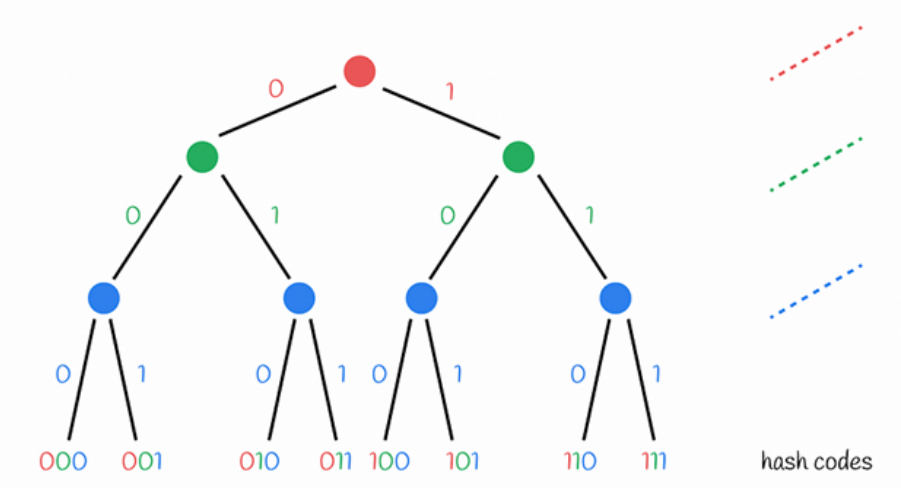
❖ Other honorable attempts:

LSH:

LSH with Random Forest Projection

Description:

- The random projections method is sometimes called **LSH Tree**. This is due to the fact that the process of hash code assignment can be represented in the form of the decision tree where each node contains a condition of whether a vector is located on the negative or positive side of a hyperplane.

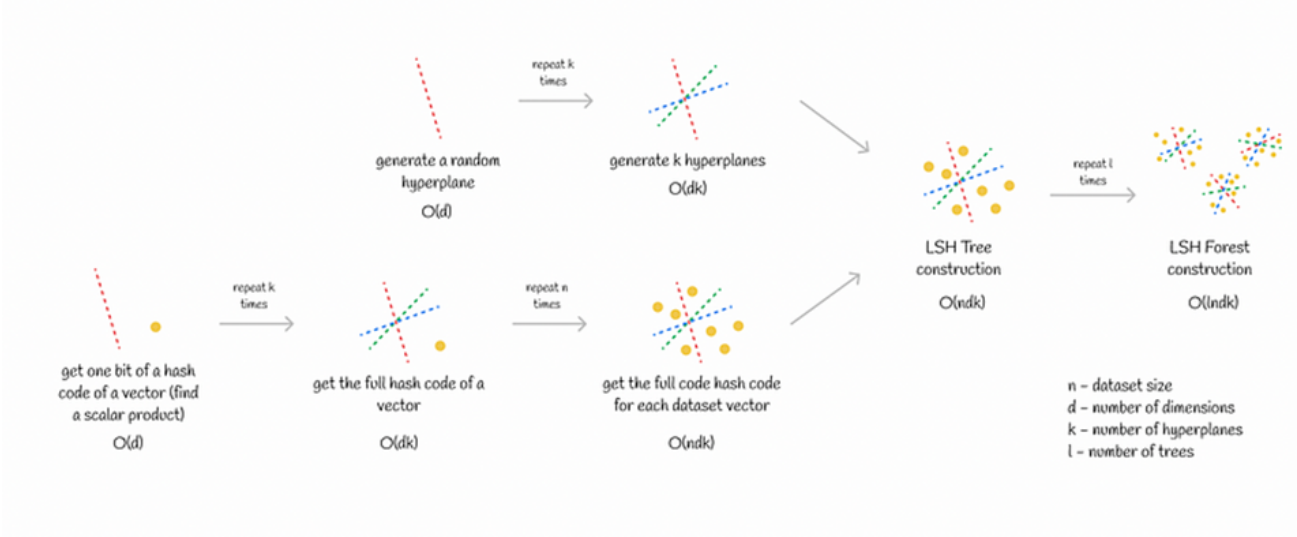


Training:

The LSH Forest training phase consists of two parts:

1. Generation of k hyperplanes. This is a relatively fast procedure since a single hyperplane in d-dimensional space can be generated in (k= 16 in 1M).
2. Assigning hash codes to all dataset vectors. This step might take time, especially for large datasets. Obtaining a single hash code requires of time. If a dataset consists of n vectors, then the total complexity becomes.

The process above is repeated several times for each tree in the forest. (forest = 20 in 1M)

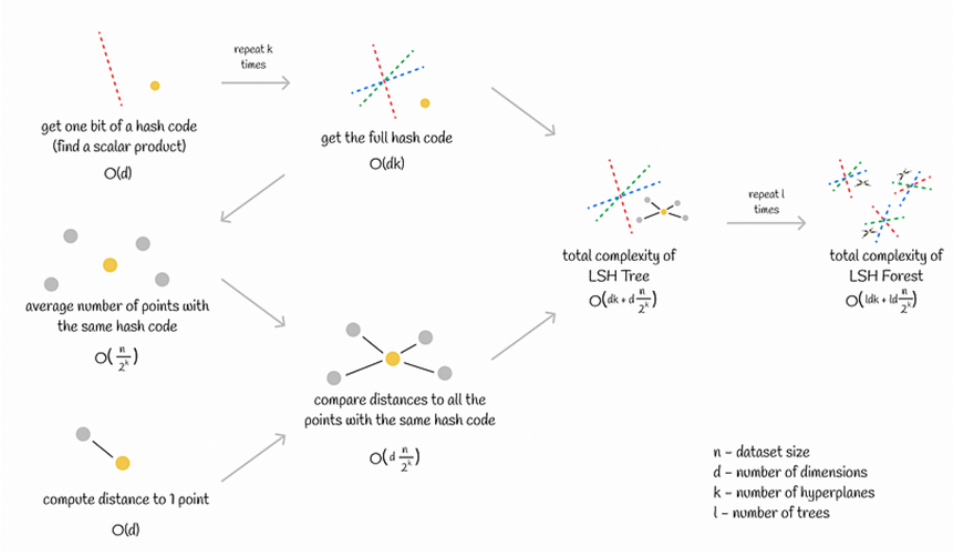


Retrieve:

One of the advantages of LSH forest is its fast inference which includes two steps:

1. *Getting the hash code of a query*. This is equivalent to computing *k* scalar products which result
2. *Finding the nearest neighbors* to the query within the same bucket

As usual, the process above is repeated several times for each tree in the forest.



Why didn't we choose it:

1. Because it consumes more RAM than IVF, Because of buckets don't have the same vectors because uniform planes are randomly generated.
2. Time is nearest to IVF (not different and sometimes IVF better)

Probing LSH

Is a technique that utilizes the idea of propagation to enhance the accuracy and efficiency of Locality Sensitive Hashing (LSH). Same as LSH random projection but in retrieval it tries to change some bits of hash code to take more buckets in calculations.

Why didn't we choose it:

1. Because it consumes more RAM and lowers accuracy since it randomly selects the buckets

PQ

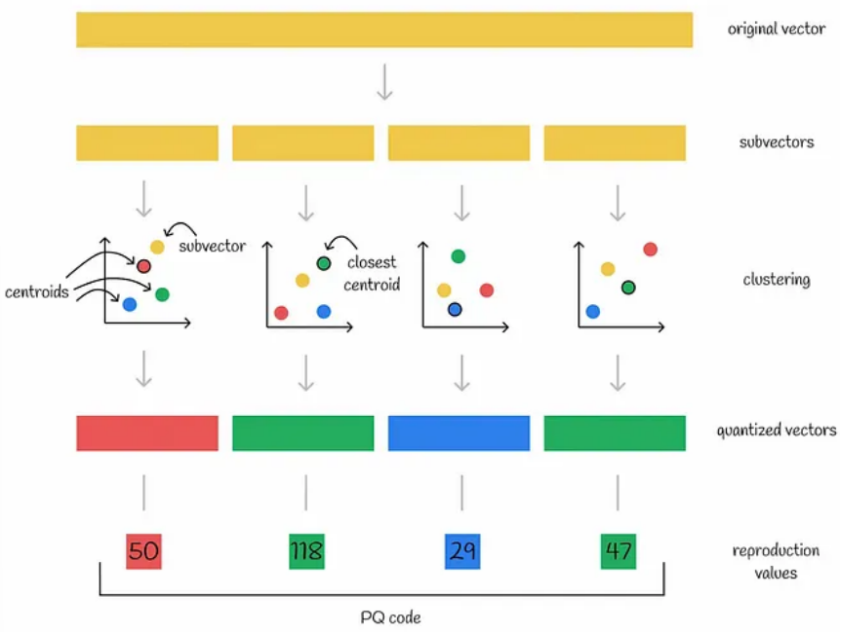
Product Quantization

Description:

- It's a compression algorithm that is used to compress the original vector to a smaller one, It's done by dividing each vector into equal parts as each of them is called subvector, each of them has an equal size for example as our vector's length is 70 then we will divide it into 14 segment (14 subvector), those subvectors are independent of each other as each one of them would be converted into an Id
- How do we get this Id?
  As we described above that every segment is independent from the other one, each of them would have its own Kmeans estimator, and this Kmeans estimator is trained to determine for which class this subvector should belong to, and number of clusters per each estimator is determined in the first place

- So, our parameters till now are the following
    1- Number of Subvectors that we will divide the original vector to
    2- Number of Clusters (Classes or Ids) which each subvector will be assigned to

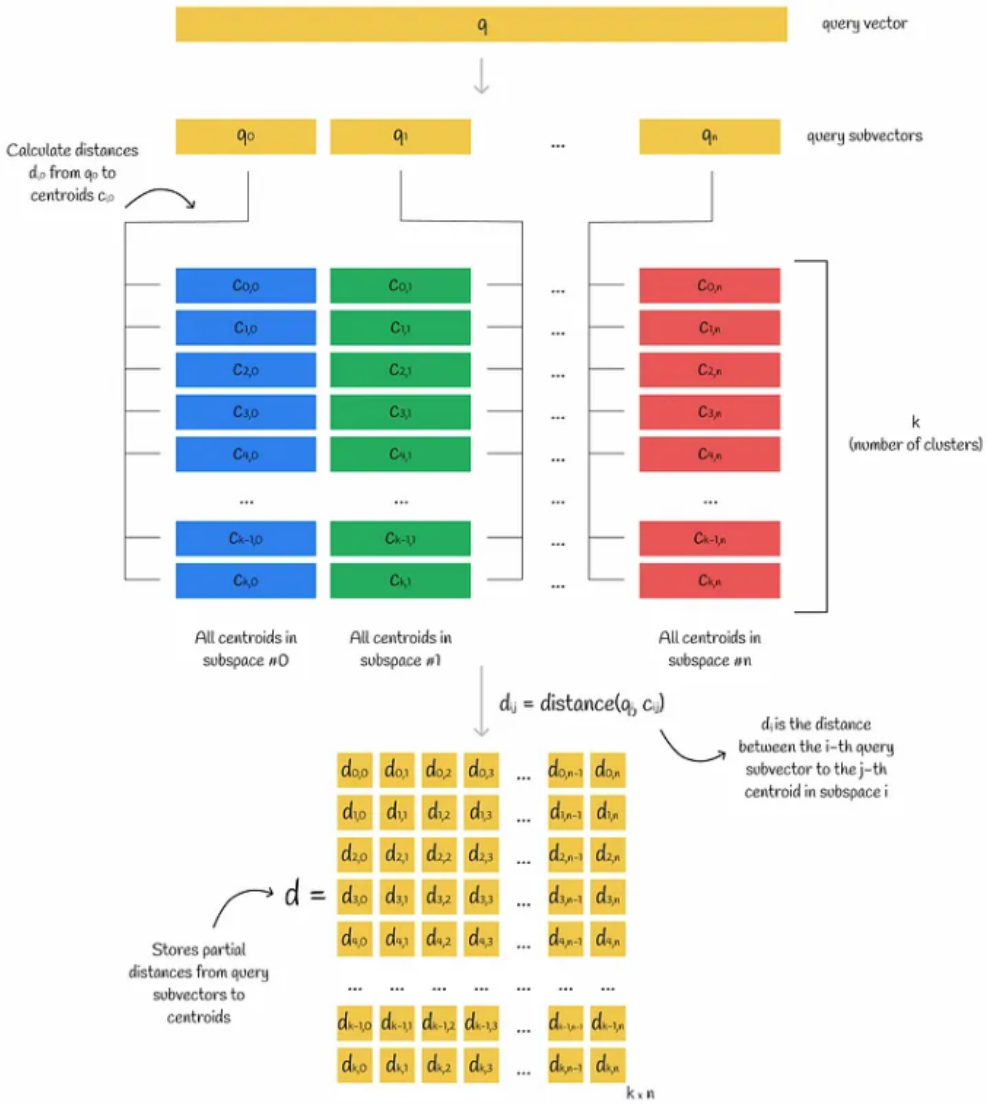
  Then we will have the vector quantized using these Kmeans into its PQ codes as the following diagram



Retrieval:

- Firstly we want to get the most similar K vectors to our query, then how do we do this?
  We need to generate a table for that query and this table would contain the partial query subvector-to centroid distances. But how to do this also?
- We will take the query vector we have and split it into segments, with the same number of segments as we set before
- Then assume we got the first segment, as we discussed earlier that each segment has its own Kmeans estimator which means that it has its own independent Centroids, then we will take the euclidean distance between the segment of the query vector and all the centroids generated from the Kmeans estimator corresponding to this segment, and this shall be the first column of our matrix, then we will keep looping calculating the distances between each segment and the corresponding centroids of the estimator corresponding to that segment

- Then we will have a matrix with size (Number of clusters , Number of segments) it's declared in the following graph



- This table is considered as a pre-setup before getting the similarity between the query and database vectors. Currently we want to compare a db vector with this query after generating this table. Then how to do this?
- the db vectors as declared above is stored in a compressed version, as it's stored with its PQ codes, then those codes are describing the id of the cluster they are referenced to, so in our table we have just generated they are describing which row assuming we know the segment (which is describing which column)
- So we would access the table we have just generated using the segments of the database vector, looping over each segment (so we know our column) then looking at this segment, our database vector is closer to which cluster (so we know our row) then get the distance from the distance table (and square it) then we shall sum all of these distances together and get the root of them then we will get the similarity between the query and this database vector as the following

- By looping over all the vectors in the database we shall get the closest vectors to our query by computing the distances between them and the query itself, and this approximation is called asymmetric distance

Results:

Data Size	Time Taken	Recall
10K	0.0632159453	0.0
100K	0.095631	0.0
1M	0.429055213	0.0

Why didn't we choose it:

- it was made in the first place to merge it with the IVF Module to create PQ-IVF combination to achieve the best accuracy and minimizing the amount of RAM used as much as possible so the module itself won't be the best option to work with only as it's looping over all the database and compare with all database vectors

Inverted File Index with Product Quantization (IVFPQ)

We have also attempted to blend PQ with IVF

1. Train the PQ using the data we got, (We took only the first 100000 record to train), and by this way we would have the independent segments centroids
2. We use the data itself to generate the centroids of the IVF and categorize each record that this one belongs to this centroid, applying the same way explained above in the IVF
3. Currently we have the centroids of IVF and the data itself, but the difference between IVFPQ and IVF is that the stored data is compressed using the PQ as explained in the PQ, by splitting it into segments and convert each segment to its corresponding PQ codes by that way we would decrease the size of the stored vectors for example from 70 float(5600 byte) in the vector into 14 integer(120 byte)
4. Then we will get a query to search for its closest vectors, then we will go through the first stage of retrieval of IVF which will guide us to the correct clusters to search in
5. But now how to search in those compressed vectors? we will use the same way of retrieval of PQ as for this query we will generate the query table, then we will loop over all the vectors the IVF sent us to and compare them with the query by calculating the asymmetric distance as we described in PQ

Then we will get the top K vectors from them to our query.

A sample result we got using IVFPQ was

1M score -100 time 0.7 RAM 2.5 MB

It was sometimes lower than this but clearly, the IVF got better recall & speed and that's why we didn't continue with the IVFPQ approach.
It was a good try to optimize the memory usage, and actually it did. The RAM usage was 2.5 MB for IVFPQ and 20 MB for IVF (Both under the max allowed RAM so that's why this wasn't a critical difference unlike speed and recall).