To add codeblocks refer : https://www.techrepublic.com/article/how-to-add-a-code-block-to-google-documents-with-the-code-blocks-extension/

**Introduction**

With the increased adoption of LLMs, vector databases are rapidly gaining more popularity with each passing day. The principal reason behind this is the speed and efficiency with which these databases can store and process vectors. Vector indexing is one of the major factors behind this speed and efficiency. This micropaper does a deep dive into various algorithms that are used for vector indexing along with code implementation. Moreover the usage of these algorithms in open source vector databases like Milvus, Qdrant, Weviate etc is also analyzed

**Need for Vector Databases and Similarity Search**

Before going into algorithms, it is important to understand why a vector database is needed. Vector databases help perform similarity search efficiently. Similarity search at its core enables us to compare data quickly. Be it text, audio, images or genome sequences, everything can be represented as vectors and compared. Some of the many application of similarity search include:

* Searching for text/documents which are most similar to a search term such as in Google
* Find similar customers so as to make new product recommendations based on the purchase history of similar customers such as Amazon (similarly in Spotify)
* Find similar genes in genome databases to help identify a particular disease
* Deduplication of datasets

**Brute Force Method aka Flat Index**

With the need for similarity search established, one of the easiest approaches to compare vectors is to go for brute force method, where a given vector is compared with all other vectors in the database. A simple Python based implementation of this approach is shown below

| import numpy as np data = np.random.normal((1000, 50)) query = np.random.normal((50,)) nearest\_index = np.argmin(np.linalg.norm(query - data, axis=1))  nearest\_vector = data[nearest\_index, :] |
| --- |

TODO : Implementation using cosine similarity

In practice, FAISS (Facebook AI Similarity Search) is used whenever these algorithms have to be used in production. FAISS is a library which comes with built in algorithms to perform similarity search very efficiently. The implementation of FlatIndex using FAISS would be as follows

| import faiss  index = faiss.IndexFlatL2(d) index.add(xb) D, I = index.search(xq, k) |
| --- |

In the above code, d stand for dimension of the vectors, xb is the collection of vectors, xq is the query vector, k is the number of nearest vectors we want the library to return, I is the list of indexes of the nearest vectors and D is their corresponding distances from the query vector

While this approach works well when the number of vectors are less, when the size grows, this approach takes too long, as the algorithm would have to do millions of comparisons

**Vector Search Algorithms**

To make vectors search faster when having to deal with millions and billions of vectors, there are multiple algorithms available. This algorithms can be broadly classified into following categories:

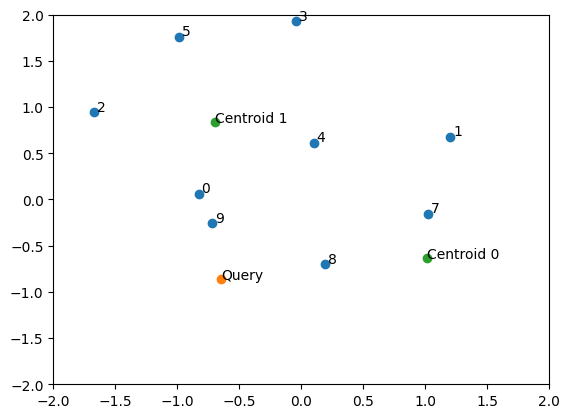
* Cluster based indexing such as IVF and PQ
* Tree based indexing such as ANNOY
* Graph based indexing such as HNSW
* Hash based indexing such as LSH

In this micropaper, the Inverted File Index (IVF), Product Quantization (PQ) and Hierarchical Navigable Small World (HNSW) algorithms will be analyzed in detail

**Inverted File Index (IVF)**

The basic principle behind this algorithm is to search only a subset of vectors which are most likely to be close to the query vector, rather than search all the vectors. To achieve this, the entire dataset of vectors is divided into partitions, with every partition having a centroid. For a given query vector, the nearest partition or cluster is found by comparing its similarity with all the centroids. The search is then restricted to only all the vectors in the nearest cluster. A simple Python based implementation is shown below:

| import numpy as np from scipy.cluster.vq import kmeans2  np.random.seed(145)  data = np.random.normal(size = (10, 2)) query = np.random.normal(size = (2, )) num\_part = 2 # number of partitions into which dataset is divided (centroids, assignments) = kmeans2(data, num\_part, iter=5) nearest\_centroid\_index = np.argmin(np.linalg.norm(query - centroids, axis=1)) min\_dis = np.inf nearest = -1 # index of nearest vector  for index, assignment in enumerate(assignments):  # search for vectors in nearest partition  if assignment == nearest\_centroid\_index:  dis = np.linalg.norm(query - data[index])  # print(dis)  if dis < min\_dis:  min\_dis = dis  nearest = index  Note that while IVF helps improves speed, it compromises on accuracy. This can be shown with below example. On running the code above the vector with index 8 is shown as the nearest vector. Whereas the actual vector which is nearest is the one with index 9. This occurs because, centroid 0 is closer to query vector than centroid 1, hence the search is restricted to only those vectors which belong to cluster 0. Since 8 is closest to query vector amongst all vectors in cluster 0, it is returned as the nearest vector. 9 belongs to cluster 1 and hence was not considered. |
| --- |



The implementation of IVF using the FAISS library would be as follows:

| import faiss  nlist = 128 # number of cells/clusters to partition data into  quantizer = faiss.IndexFlatIP(d) # how the vectors will be stored/compared index = faiss.IndexIVFFlat(quantizer, d, nlist) index.train(xb) # we must train the index to cluster into cells index.add(xb) index.nprobe = 8 # set how many of nearest cells to search  D, I = index.search(xq, k) |
| --- |

In the above code, nlist stands for number of clusters (or inverted lists) to partition data into, d stand for dimension of the vectors, xb is the collection of vectors, nprobe,tells how many of the nearest clusters to search, xq is the query vector, k is the number of nearest vectors we want the library to return, I is the list of indexes of the nearest vectors and D is their corresponding distances from the query vector.

Note that unlike FlatIndex, the IVFIndex needs to be trained as clusters have to be formed based on the data. Moreover, search can be extended to multiple clusters which are near, instead of only searching in only the nearest cluster. This is especially useful when the dimensionality of vectors is high.

TO DO : Add more optimized code

**Scalar Quantization**

Product quantization uses some of the principles of scalar quantization, hence scalar quantization is discussed briefly. Scalar quantization in vector search engines is a compression technique that compresses vectors by reducing the number of bits used to represent each vector component. To do so, takes the maximum and minimum value of a particular dimension as seen across all the vectors, and uniformly splits that dimension into bins across its entire range. The Python implementation of this would be as follows:

| import numpy as np  np.random.seed(145)  data = np.random.normal(size=(4, 2))  data\_quantized = np.zeros(shape=data.shape) for i in range(data.shape[1]):  min = data[:, i].min() # get min of dimension  max = data[:, i].max() # get max of dimension  data\_quantized[:, i] = np.uint(((data[:, i] - min) \* 255) / (max - min)) |
| --- |

Lets print the data pre and post quantization to see the difference

| >>> data array([[-0.82862278, 0.06038149],  [ 1.20277692, 0.67641474],  [-1.66968675, 0.9454519 ],  [-0.04539016, 1.92633477]]) >>> data\_quantized array([[ 74., 0.],  [255., 84.],  [ 0., 120.],  [144., 255.]]) |
| --- |

It can be observed how for each dimension (i.e. column), the values have been mapped from 0 to 255, with the minimum value being mapped to zero and maximum being mapped to 255. The values in between are mapped by uniformly splitting the difference between the min and max values into 255 bins and mapping the value to the closest bin. For example, in the first dimension, the difference between -1.669 (min) and 1.202 (max) i.e. 2.871 is split into 255 bins, so the size of each bin is approximately 0.0112. The 73rd bin corresponds to -0.828 and 74th bin to -0.8402, hence all values in between this get mapped to 74. The same works for the rest of the values.

The same code can be written more efficiently as follows:

| import numpy as np  np.random.seed(145)  data = np.random.normal(size=(4, 2))  data\_quantized = np.zeros(shape=data.shape) ranges = np.vstack((np.min(data, axis=0), np.max(data, axis=0))) starts = ranges[0,:] steps = (ranges[1,:] - ranges[0,:]) / 255 data\_quantized = np.uint8((data - starts) / steps) |
| --- |

By using scalar quantization, each element can be represented by 1 byte (i.e. size of int8) instead of 8 bytes(i.e. size of float64), thus reducing the amount of memory required to store a vector by 8x, as shown below

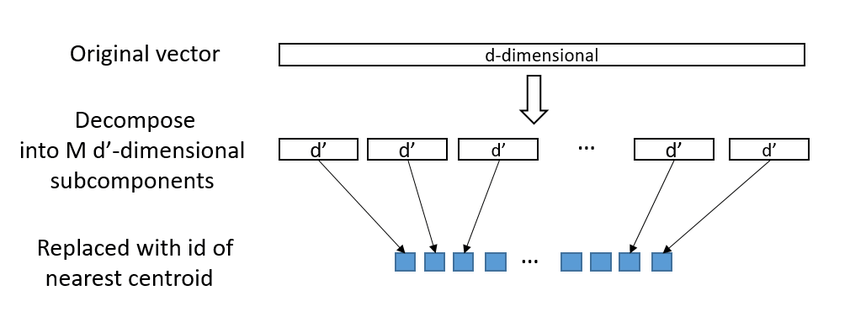
| >>> data.nbytes 64 >>> data\_quantized.nbytes 8 |
| --- |

Note: In the example above, *data* variable has 4 vectors each of length 2 hence a total of 8 elements. Each element occupies 8 bytes (since they are float64), hence the total size of variable is 8 x 8 = 64 bytes.

When there are millions and billions of vectors, this can help significantly reduce the amount of memory required. The drawback of this process is loss of accuracy.

**Vector and Product Quantization**

Scalar quantization quantizes each dimension separately, whereas vector quantization quantizes groups of dimensions at once, hence the name vector quantization.

Product quantization is a method of compressing vectors to minimize their memory usage by dividing them into chunks and quantizing each vector segment individually. Each chunk, or more specifically, each vector in the lower dimensional subspace is approximated by the centroid of the cluster it belongs to within the subspace. The positions of the centroids are determined through the utilization of a clustering algorithm such as k-means. Each centroid is assigned an unique index, called a centroid index. Once centroids are computed, we replace all subvectors in the original dataset with the ID of its closest centroid. The original vector is thus represented by a collection of IDs, each ID corresponding to a centroid in that particular subspace.

The algorithm is explained below with an example

Assume there are 9 vectors, each of length 4.

| np.random.seed(145) data = np.random.normal(size=(9, 4)) |
| --- |

**Step 1** : Each vector is divided into K subvectors (here K=2). The size of each subvector (aka sublen) would be d / K = 4 / 2 = 2.The zeroth segment (i.e. from element 0 to sublen) of all the vectors forms a subspace, the first segment of all the vectors forms another subspace

| K = 2 # Find length of each subvector by dividing total length by number of subvectors # In this example 4/2 = 2 is the length of each subvector sublen = data.shape[1] // K # To get zeroth subspace zeroth\_subspace = data[:, 0:sublen] array([[-0.82862278, 0.06038149],  [-1.66968675, 0.9454519 ],  [ 0.09937902, 0.61274873],  [ 1.64287431, -2.34978337],  [ 0.19061736, -0.7009628 ],  [-0.64845217, -0.85808538],  [-0.55737482, -1.19450904],  [ 1.46979984, -0.79896093],  [ 0.45038609, 0.33732225]]) |
| --- |

**Step 2** : Compute centroids for each subspace. Each centroid is given an id. The example below shows this for zeroth subspace. Centroid ids are stored in assignment variable

(centroids, assignments) = kmeans2(zeroth\_subspace, K, iter=4)

Since there are 2 clusters, centroids ids are 0 and 1 (each id representing a vector)

array([1, 1, 1, 0, 0, 0, 0, 0, 1], dtype=int32)

**Step 3**: The ids are stored to represent the final quantized vector

quantized\_dataset[:, 0] = np.uint8(assignments)

These steps are repeated across all the subspaces. The final quantized output in this example would be

| array([[1, 0],  [1, 2],  [1, 2],  [2, 0],  [0, 1],  [0, 1],  [0, 1],  [2, 0],  [1, 0]], dtype=uint8) |
| --- |

The final code would be

| import numpy as np from scipy.cluster.vq import kmeans2   class ProductQuantizer:   def \_\_init\_\_(self, M=2, K=3):  self.M = M # number of subvectors  self.K = K # number of clusters within each subspace  self.\_dataset = None   def create(self, dataset):  """Fits PQ model based on the input dataset."""  sublen = dataset.shape[1] // self.M # length of each subvector  self.\_centroids = np.empty((self.M, self.K, sublen), dtype=np.float64)  self.\_dataset = np.empty((dataset.shape[0], self.M), dtype=np.uint8)  for m in range(self.M):  subspace = dataset[:,m\*sublen:(m+1)\*sublen]  (centroids, assignments) = kmeans2(subspace, self.K, iter=32)  self.\_centroids[m,:,:] = centroids  self.\_dataset[:,m] = np.uint8(assignments)  print(self.\_dataset)   def quantize(self, vector):  """Quantizes the input vector based on PQ parameters"""  quantized = np.empty((self.M,), dtype=np.uint8)  for m in range(self.M):  centroids = self.\_centroids[m,:,:]  distances = np.linalg.norm(vector - centroids, axis=1)  quantized[m] = np.argmin(distances)  return quantized   def restore(self, vector):  """Restores the original vector using PQ parameters."""  return np.hstack([self.\_centroids[m,vector[m],:] for m in range(self.M)])   @property  def dataset(self):  return self.\_dataset |
| --- |

Note that while it looks as if the vectors have undergone dimensionality reduction, that is not the case. Each number/index in the above example represents a 2 dimensional vector, hence every vector is still a 4 dimensional vector. This can be checked using the restore function

| >>> quantizer.restore([1, 2]) array([ 1.46979984, -0.79896093, -0.36128112, -1.25485514]) |
| --- |

Product quantization can cause significant loss in accuracy (much more than scalar quantization) and is slower as well. PQ is recommended only if the memory footprint is the top priority and the search speed is not critical. It is generally used for high dimensional vectors.

**Dimensionality Reduction vs Quantization**

Note that quantization is not the same as dimensionality reduction. In dimensionality reduction, the length of each vector is reduced. In quantization the length of the vector remains unchanged, but the precision is reduced. Moreover dimensionality reduction requires the data distribution to follow certain criterias (for example PCA requires data that can be separated into independent, Gaussian distributed components) but quantization makes no assumption regarding the distribution of the data.

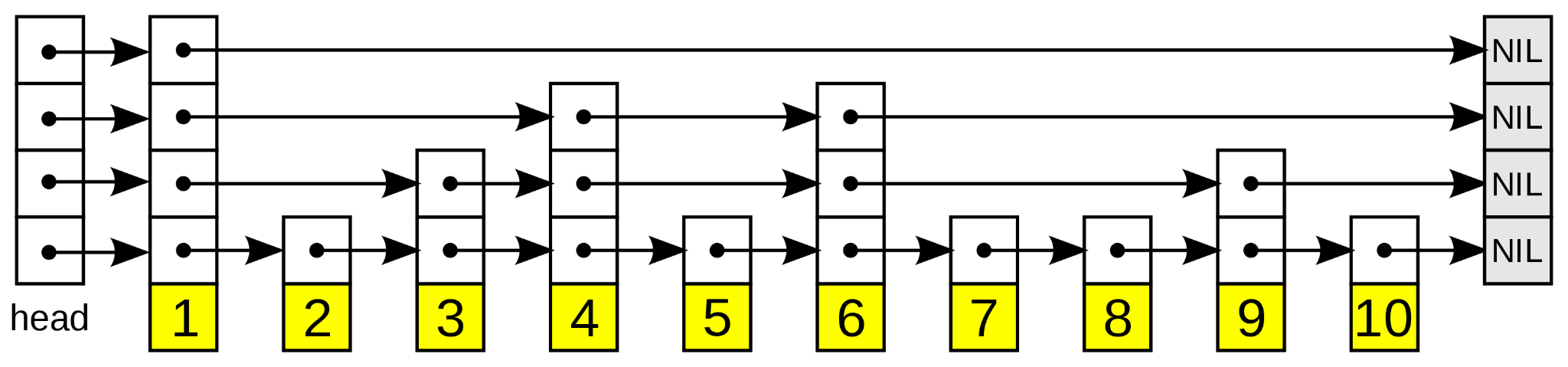
**HNSW**

HNSW is one of the most popular and commonly used algorithms when it comes to vector indexing. HNSW is based on 2 core concepts — skip list and navigable small world.

**Skip List**

Linked list is a data structure. It consists of nodes where each node contains data and a reference to the next node in the sequence (this reference is required because data items in a linked list are not stored contiguously in memory or on disk). One major issue with linked list is to to access any random element is O(n). Skip lists solve this issue by introducing additional layers that help reduce time to access to O(log n). The working of the algorithm is explained with an example below

As shown below skip list is a multi-level linked list where the upper levels maintain long connections while the lower level maintain shorter connections.Once we find a node corresponding to an element in the list greater than i, we backtrack to the previous node and move to the layer below. This continues until we've found the element we're looking for. Note that skip lists only work for sorted lists, as we need a way to compare the magnitude of two objects directly



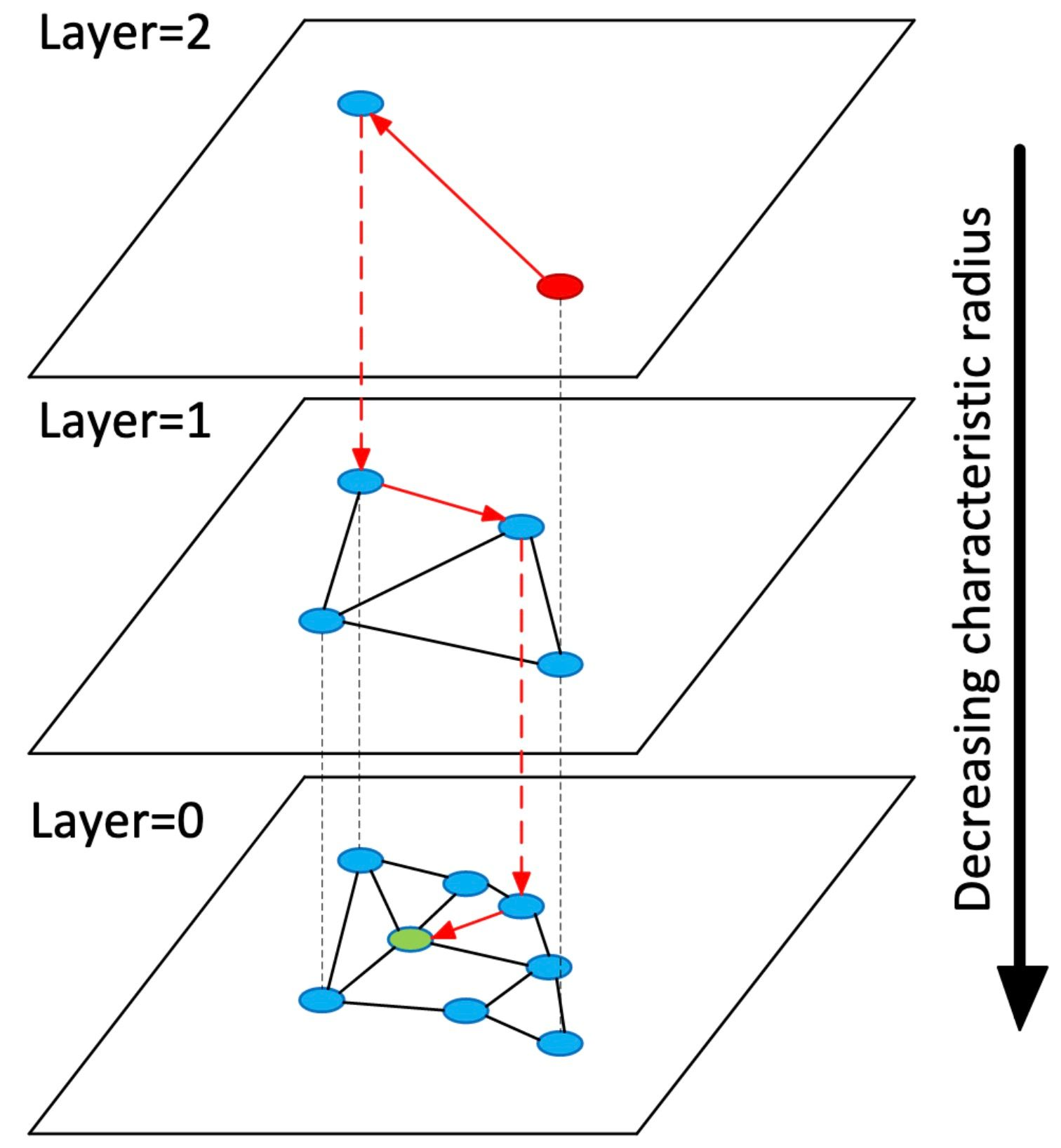
For example, assume we want to search number 5. At the top most level, we start from 1 and it directly goes to NIL, so we backtrack to 1 and go one level down. In this second level, we go to the next element 4, which is less than 5. So we again go to the next element, which here is 6. Since 6 > 5, it goes back to 4 and goes down to the 3rd layer. In the 3rd layer it again goes forward to 6, backtracks since it is greater than 5, and goes to the 4th layer. In the 4th layer, the pointer goes to 5 and the search ends as the element is found.

In a way similar to binary search, with the difference that instead of search on both sides of the target, search is only on one side

**HNSW**

Like the skip list, HNSW maintains multiple layers of NSWs. The uppermost layer of an HNSW graph has few nodes and the longest links, while the bottommost layer has all nodes and the shortest links. During the search process, algorithm starts from a pre-defined point in the uppermost layer and greedily routes toward the nearest neighbor to the query vector.

TO DO : Code explanation



The code implementation would be as follows

| from bisect import insort from heapq import heapify, heappop, heappush  import numpy as np  from .\_base import \_BaseIndex   class HNSW(\_BaseIndex):   def \_\_init\_\_(self, L=5, mL=0.62, efc=10):  self.\_L = L  self.\_mL = mL  self.\_efc = efc  self.\_index = [[] for \_ in range(L)]   @staticmethod  def \_search\_layer(graph, entry, query, ef=1):   best = (np.linalg.norm(graph[entry][0] - query), entry)   nns = [best]  visit = set(best) # set of visited nodes  candid = [best] # candidate nodes to insert into nearest neighbors  heapify(candid)   # find top-k nearest neighbors  while candid:  cv = heappop(candid)   if nns[-1][0] > cv[0]:  break   # loop through all nearest neighbors to the candidate vector  for e in graph[cv[1]][1]:  d = np.linalg.norm(graph[e][0] - query)  if (d, e) not in visit:  visit.add((d, e))   # push only "better" vectors into candidate heap  if d < nns[-1][0] or len(nns) < ef:  heappush(candid, (d, e))  insort(nns, (d, e))  if len(nns) > ef:  nns.pop()   return nns   def create(self, dataset):  for v in dataset:  self.insert(v)   def search(self, query, ef=1):   # if the index is empty, return an empty list  if not self.\_index[0]:  return []   best\_v = 0 # set the initial best vertex to the entry point  for graph in self.\_index:  best\_d, best\_v = HNSW.\_search\_layer(graph, best\_v, query, ef=1)[0]  if graph[best\_v][2]:  best\_v = graph[best\_v][2]  else:  return HNSW.\_search\_layer(graph, best\_v, query, ef=ef)   def \_get\_insert\_layer(self):  # ml is a multiplicative factor used to normalize the distribution  l = -int(np.log(np.random.random()) \* self.\_mL)  return min(l, self.\_L-1)   def insert(self, vec, efc=10):   # if the index is empty, insert the vector into all layers and return  if not self.\_index[0]:  i = None  for graph in self.\_index[::-1]:  graph.append((vec, [], i))  i = 0  return   l = self.\_get\_insert\_layer()   start\_v = 0  for n, graph in enumerate(self.\_index):   # perform insertion for layers [l, L) only  if n < l:  \_, start\_v = self.\_search\_layer(graph, start\_v, vec, ef=1)[0]  else:  node = (vec, [], len(self.\_index[n+1]) if n < self.\_L-1 else None)  nns = self.\_search\_layer(graph, start\_v, vec, ef=efc)  for nn in nns:  node[1].append(nn[1]) # outbound connections to NNs  graph[nn[1]][1].append(len(graph)) # inbound connections to node  graph.append(node)   # set the starting vertex to the nearest neighbor in the next layer  start\_v = graph[start\_v][2] |
| --- |

The implementation of HNSW using FAISS library is as follows

| import faiss M = 64 # number of connections each vertex will have ef\_search = 32 # depth of layers explored during search ef\_construction = 32 # depth of layers explored during index construction  # initialize index (d == 128) index = faiss.IndexHNSWFlat(d, M) # set efConstruction and efSearch parameters index.hnsw.efConstruction = ef\_construction index.hnsw.efSearch = ef\_search # add data to index index.add(xb) D, I = index.search(xq, k) |
| --- |

TO DO : Explain variables in HNSW FAISS code

**Comparison of different algorithms**

In this section, the different algorithms discussed in the previous sections would be compared wrt speed and accuracy. The following two metrics would be used for comparison

* Recall@1 : In information retrieval, recall is Total number of documents retrieved that are relevant divided by Total number of relevant documents in the database. Recall@1 calculates if the nearest vector given by algorithm is equal to the actual ground truth nearest vector (this is done across all the vectors)
* Time for search : Time taken to find the k nearest vectors for all the vectors in database

The evaluation function to compute these 2 metrics is shown below

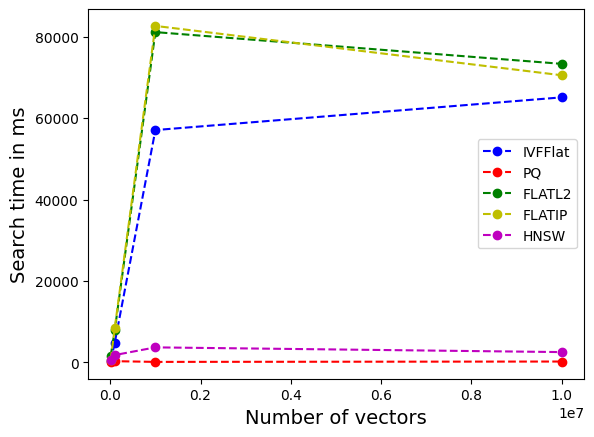
| def evaluate(index):  import time  # for timing with a single core  # faiss.omp\_set\_num\_threads(1)  t0 = time.time()  D, I = index.search(xq, k)  #print(I)  t1 = time.time()    recall\_at\_1 = (I[:,:1] == gt[:, :1]).sum() / float(nq)  print("\t %7.3f ms, R@1 %.4f, missing rate %.4f" % (  (t1 - t0) \* 1000.0, recall\_at\_1, missing\_rate))  return (t1 - t0) \* 1000.0 , recall\_at\_1 |
| --- |

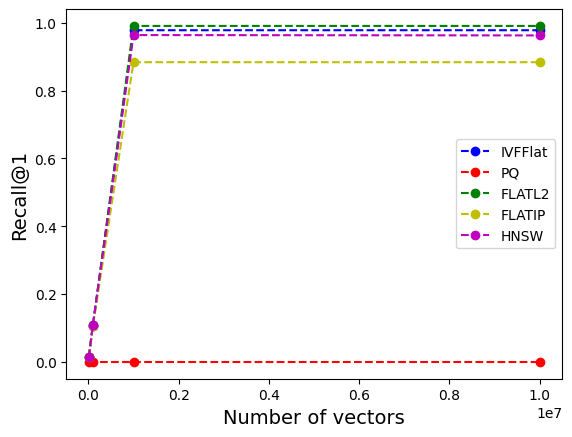
In the code above, gt is the ground truth (i.e. the actual nearest vectors indices), I is the indices of the nearest vectors given by the algorithm, nq is the query vector.

|  | **10k** | **100k** | **1M** | **10M** |
| --- | --- | --- | --- | --- |
| **ivfflat** | 264.447 | 4824.841 | 57064.866 | 65096.212 |
| **pq** | 97.296 | 279.706 | 105.25 | 191.904 |
| **flatl2** | 1555.469 | 7866.423 | 81129.343 | 73335.25 |
| **flatip** | 675.846 | 8464.573 | 82651.069 | 70534.024 |
| **hnsw** | 557.919 | 1783.802 | 3671.637 | 2513.107 |

|  | **10k** | **100k** | **1M** | **10M** |
| --- | --- | --- | --- | --- |
| **ivfflat** | 0.014 | 0.1072 | 0.9783 | 0.9783 |
| **pq** | 0.0 | 0.0 | 0.0 | 0.0 |
| **flatl2** | 0.014 | 0.109 | 0.991 | 0.991 |
| **flatip** | 0.014 | 0.105 | 0.884 | 0.884 |
| **hnsw** | 0.014 | 0.109 | 0.964 | 0.963 |

The above data can be visualized as shown below

****

****

From the 2 graphs, it is clear that while HNSW and PQ are significantly faster than the other algorithms, PQ has very poor accuracy whereas HNSW has very high accuracy, almost comparable with FlatIndex. This is the reason why most vector databases prefer to use HNSW as their vector indexing algorithm, as explored in the next section.

Note: As of now, the authors were not able to figure out why the time remained nearly constant when the number of vectors increased from 1 million to 10 million. Analyzing this in more detail would be part of future work.

**When to use which algorithm**

https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index

Refer FAISS Wiki and Zilliz

**Application of algorithms in Open Source Vector databases**

TO DO : Difference b/w vector library and vector database

Before going into vector databases, it is important to understand the difference between a vector library and a vector database

In this section we go through various open source vector databases and look at which algorithms they use for vector indexing.

**Milvus**

Milvus is an open source vector database built using Go and Python. It supports the following algorithms:

* Flat
* IVF Flat
* IVF PQ
* HNSW

and a few more.

**Qdrant**

Qdrant is an open source vector database built using Rust. It only supports HNSW as a dense vector index. It also offers scalar and vector quantization an optional feature to enable efficient storage of high dimensional vectors

**Weviate**

Weviate is an open source vector database built using Go. It supports two types of vector indexing:

* Flat
* HNSW

**Elastic Search**

ElasticSearch is an open source, distributed, RESTful search engine built using Java on top of Apache Lucene. Unlike the previous 3 solutions mentioned above, ElasticSearch is not a vector native solution i.e. while the other 3 have been built ground up for vector search, ES was initially built as a search engine for searching text, structured and unstructured data, with vector search as a feature being added later.

ES currently supports the HNSW algorithm

Apart from the solutions mentioned above, there are other closed source solutions like Pinecone, MongoDB Atlas which have not been analyzed.

**Conclusion and Future Work**

Thus in this micropaper different vector indexing algorithms were analyzed in great depth, and performance was compared. Moreover, the usage of these algorithms in multiple open source vector bases was also discussed.

Future work includes understanding how libraries like FAISS internally implements these algorithms and the various optimizations performed by these libraries such that even when the number of vector increases significantly the search time remains nearly constant

**References**

1. [**https://milvus.io/docs/v2.0.x/index.md**](https://milvus.io/docs/v2.0.x/index.md)
2. [**https://qdrant.tech/documentation/concepts/indexing**](https://qdrant.tech/documentation/concepts/indexing)
3. [**https://qdrant.tech/documentation/guides/quantization/**](https://qdrant.tech/documentation/guides/quantization/)
4. [**https://weaviate.io/developers/weaviate/concepts/vector-index**](https://weaviate.io/developers/weaviate/concepts/vector-index)
5. [**https://www.elastic.co/search-labs/blog/vector-search-elasticsearch-rationale**](https://www.elastic.co/search-labs/blog/vector-search-elasticsearch-rationale)
6. [**https://redblink.com/pinecone-vs-elasticsearch**](https://redblink.com/pinecone-vs-elasticsearch)

**Comparison of different vector databases**

**https://zilliz.com/comparison/qdrant-vs-faiss**