A deep dive into flat and IVF indexing strategies.

@Milvus.io

Vector Database 101 - Vector Index Basics and the Inverted File Index

In the previous tutorial, we went over a quick word embedding example to better understand the power of embeddings along with how they are stored and indexed in a vector database. This led to a brief overview of nearest neighbor search algorithms, a computing problem that involves finding the closest vector(s) to a query vector based on a selected distance metric. We then briefly discussed a couple of well-known methods for vector search: Vector search is a critical component of vector databases, but only as it relates to computation; vector databases are complex beasts that involve numerous moving components and layers of abstraction.

In this tutorial, we'll analyze the components of a modern indexer before going over two of the simplest and most basic indexing strategies - flat indexing and inverted file (IVF) indexes. Knowledge of these two index types will be critical as we progress into the next couple of tutorials.

Vector indexes in Milvus

Milvus uses Facebook AI Similarity Search (FAISS) as one of the key index-building libraries, along with Hnswlib and Annoy. As mentioned in the previous tutorial, Milvus builds on top of these libraries to provide a full-fledged database, complete with all the usual database features and a consistent user-level API.

If you're already familiar with FAISS, many of the concepts introduced here and in the next couple of tutorials may already be familar to you.

Indexing basics

You may have noticed this after going through the previous tutorial, but there are, broadly speaking, four different types of vector search algorithms: *hash-based* (e.g. locality-sensitive hashing), *tree-based* (e.g. ANNOY), *cluster-based* (e.g. product quantization), and *graph-based* (e.g. HNSW). Different types of algorithms work better for varying types of vector data, but all of them help speed up the vector search process at the cost of a bit of accuracy/recall.

One key detail that often goes overlooked with vector search is the capability to combine many vector search algorithms together. Within a vector database, a full vector index is generally composed of three distinct components: 1) an optional **pre-processing** step where vectors may be reduced or optimized prior to indexing, 2) a required **primary** step which is the core algorithm used for indexing, and 3)

an optional **secondary** step where vectors may be quantized or hashed to further improve search speeds.

Let's break this down bit by bit.

The first step simply prepares the vectors for indexing and search without actually building any data structure. The algorithm used here often depends on the application and the upstream vector generation method, but some commonly used ones include L2 normalization, dimensionality reduction, and zero padding. Most vector databases skip this step and leave pre-processing entirely up to the user in the application layer.

The primary algorithm is the only mandatory component and forms the crux of the index. The output of this step should be a data structure which holds all information necessary to conduct an efficient vector search. Tree-based and graph-based data structures are commonly used here, but a quantization algorithm such as product quantization or locality-sensitive hashing works as well.

The secondary step reduces the total size of the index by mapping all floating point values in the dataset into lower-precision integer values, i.e. `float64` -> `int8` or `float32` -> `int8`. This modification can both reduce the index size as well as improve search speeds, but generally at the cost of some precision. There are a couple different ways this can be done; we'll dive deeper into quantization and hashing in future tutorials.

Flat indexing

Before diving too deep into more complex vector search algorithms, it pays take a brief look at *linear search*, also known as "flat" indexing. Flat indexing is by and large the most basic indexing strategy, but arguably also the most overlooked. With flat indexing, we compare a query vector with every other vector in our database. In code, this would look something like this:

```
>>> query = np.random.normal(size=(128,))
>>> dataset = np.random.normal(size=(1000, 128))
>>> nearest = np.argmin(np.linalg.norm(dataset - query, axis=1))
>>> nearest
333
```

Note how the index is simply a flat data structure which is exactly the size of the dataset - no more and no less.

The first two lines of code creates a random query vector in addition to a 1000-element dataset of vectors. The third line then computes the distance (via `np.linalg.norm`) between all elements in the dataset and the query vector before extracting the index of the minimum distance (via `np.argmin`). This gives us the array index of the nearest neighbor to the query vector, which we can then extract using `dataset[nearest,:]`.

This is obviously the most naïve way to perform vector search, but it can work surprisingly well for small datasets, especially if you have an accelerator such as a GPU or FPGA to parallelize the search process on. The loop above, for example, runs in less than 0.5 milliseconds on an Intel i7-9750H CPU, which means that we can achieve a QPS of over 2000 with flat indexing on a six-core laptop CPU!

Our QPS drops to around 160 with 10k vectors and 16 with 100k vectors, with the >10x factor drop from 1000 vectors to 10k vectors likely being due to CPU cache size limitations. These numbers are still pretty good though. With all the talk nowadays about runtime complexity and horizontal scaling, remember the KISS principle for small applications and/or prototyping: Keep It Simple, Stupid.

Inverted file index

Flat indexing is great, but it obviously doesn't scale. This is where data structures for vector search come into play. By trading off a bit of accuracy/recall for improved runtime, we can significantly improve both query speed and throughput. There are a *lot* of indexing strategies out there today, but one of the most commonly used ones is *inverted file index* (IVF).

Fancy name aside, IVF is actually fairly simple. IVF reduces the overall search scope by arranging the entire dataset into partitions. All partitions are associated with a *centroid*, and every vector in the dataset is assigned to a partition that corresponds to its nearest centroid.

A two-dimensional Voronoi diagram. Image by Balu Ertl, CC BY-SA 4.0.

If you're familiar with FAISS, the above diagram might be familiar to you; it's called a *Voronoi diagram* and visually illustrates this cluster assignment, albeit in only two dimensions. There are a total of 20 cells (clusters), with the centroid for each cluster displayed as a black dot. All points in a dataset will fall into one of these 20 regions.

Cluster centroids are usually determined with a clustering algorithm called *k-means*. K-means is an interative algorithm that works by first randomly selecting a set of `K` points as clusters. At every iteration, all points in the dataset of vectors are assigned to its nearest centroid, and all centroids are then updated to the mean of each cluster. This process then continues until convergence - a process known as expectation-maximazation for folks familiar with statistics.

Armed with this knowledge, let's use k-means to "automagically" determine centroids for IVF. For this, we'll use scipy's `kmeans2` implementation:

```
>>> import numpy as np
>>> from scipy.cluster.vq import kmeans2
>>> num_part = 16  # number of IVF partitions
>>> dataset = np.random.normal(size=(1000, 128))
>>> (centroids, assignments) = kmeans2(dataset, num_part, iter=32)
>>> centroids.shape
(16, 128)
```

```
>>> indexes.shape
(1000,)
```

`centroids` now contains all `num_part` (in FAISS, this parameter is called `nlist`) centroids for our dataset, while `assignments` contains ID of the centroid/cluster that is closest to each vector. We can verify this as follows:

```
>>> test = [np.argmin(np.linalg.norm(vec - centroids, axis=1)) for vec in dataset]
>>> np.all(test == assignments)
True
```

We'll now need to create the inverted index by correlating each centroid with a list of vectors within the cluster:

```
>>> index = [[] for _ in range(num_part)]
>>> for n, k in enumerate(assignments):
...     index[k].append(n)  # the nth vector gets added to the kth cluster
...
```

The code above first creates a list of lists, with the outermost layer corresponding to the number of IVF partitions. The for loop then loops through all assignments (i.e. which partition each vector belongs to) and populates the index.

With the index in place, we can now restrict the overall search space by searching only the nearest cluster:

```
>>> query = np.random.normal(size=(128,))
>>> c = np.argmin(np.linalg.norm(centroids - query, axis=1))  # find the nearest partition
>>> nearest = np.argmin(np.linalg.norm(dataset[index[c]] - query, axis=1))  # find nearest neighbor
>>> nearest
333
```

With an `num_part` value of 16 and dataset size of 100k, we get around 150 QPS using the same hardware as before (Intel i7-9750H CPU). Bumping `num_part` to 64 nets us a whopping 650 QPS.

Note that it's often pragmatic to extend our search beyond just the nearest cluster, especially for high-dimensional data (for those familiar with FAISS, this corresponds to the `nprobe` parameter when creating an IVF index). This is largely due to the *curse of dimensionality*, where each partition has a significantly larger number of edges when compared with similar data in two or three dimensions. There's no good rule of thumb for a good value of `nprobe` to use - rather, it's helpful to first experiment with your data to see the speed versus accuracy/recall tradeoffs.

And that's it for IVF! Not too bad, right?

Wrapping up

In this tutorial, we went looked at the three individual components of a vector index along with two of the most commonly used methods - flat indexing and the inverted file index. These are two of the most basic strategies, and we'll use them as a launchpad for further deep dives into more complex indexes.

In the next tutorial, we'll continue our deep dive into indexing strategies with scalar quantization (SQ) and product quantization (PQ) - two popular quantization strategies available to Milvus users. See you in the next tutorial!

All code for this tutorial is freely available on Github: https://github.com/fzliu/vector-search.