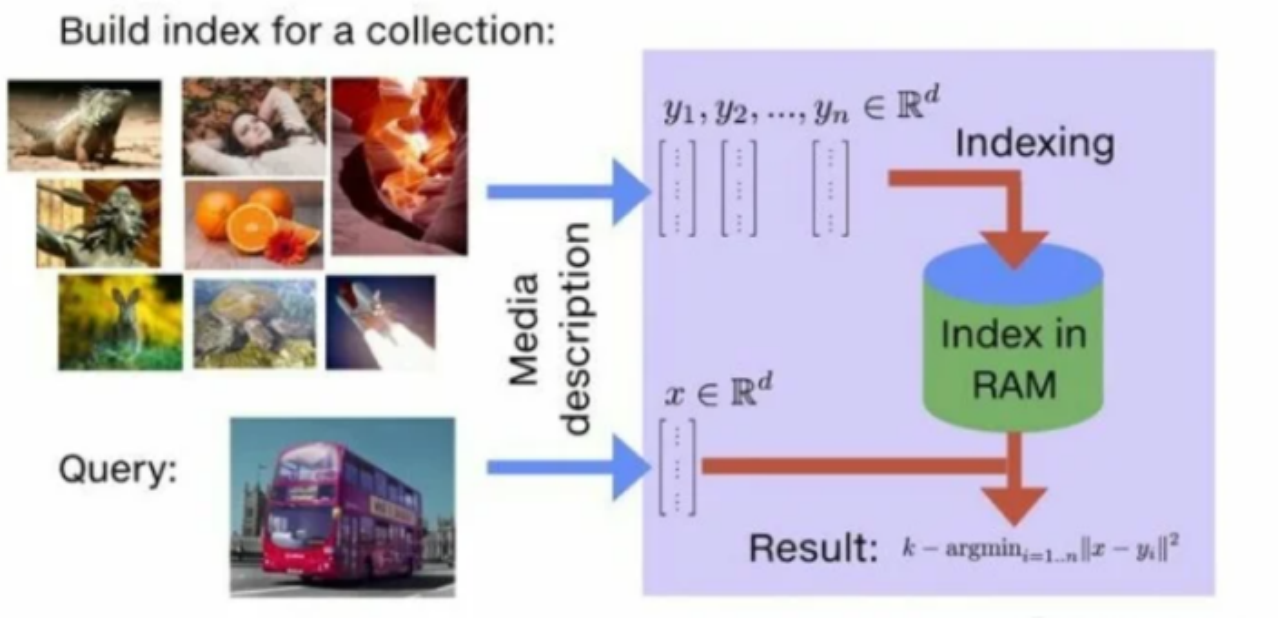
最近在使用ColBERT双塔结构进行文本召回，其中必然要涉及到向量相似度查询，如果只用brute-force方法的复杂度实在太高，无法接受，所以必须在Faiss上建立索引。因此，今天来学习一下Faiss的原理和实际应用。

在这个万物皆可embedding的时代，图像、文本、商品皆被表示为50-1000维的向量。在双塔结构中，我们把物品的embedding离线存好；当query来的时候，就要在一个巨大的商品池中去召回top k个商品(k 为千级)。这个过程必须很快的完成，来达到较大的QPS(query per second). 那么，怎么去快速召回相似向量呢？



双塔结构

Faiss是Facebook AI团队开源的针对聚类和相似性搜索库，为稠密向量提供高效相似度搜索和聚类，支持十亿级别向量的搜索，是目前最为成熟的近似近邻搜索库。它包含多种搜索任意大小向量集（向量集大小由RAM内存决定）的算法，以及用于算法评估和参数调整的支持代码。Faiss用C++编写，并提供与Numpy完美衔接的Python接口。除此以外，对一些核心算法提供了GPU实现。

1. Faiss简介

如果用暴力搜索的方法，能够得到完全正确的“标准答案”，但是其时间复杂度为O(mn)，这根本无法接受。如果牺牲一些精度的话，比如允许与参考结果有一点点偏差，那么相似性搜索能快几个数量级。加快搜索速度还涉及到数据集的预处理，我们通常把这个预处理操作称作索引。我们主要关注三个评价指标：

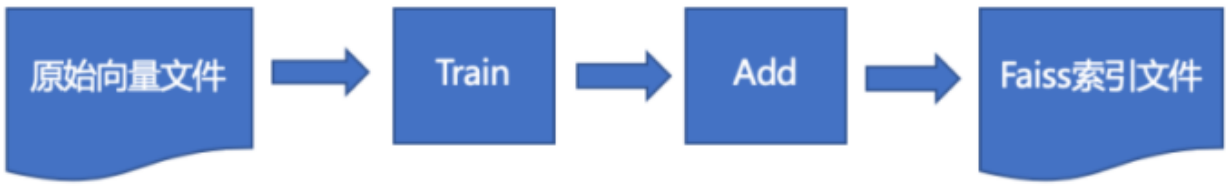
- 速度。找到与查询最相似的k个向量要耗时多久？期望比暴力算法耗时更少，不然索引的意义何在？
- 内存消耗。该方法需要消耗多少 RAM？Faiss 支持只在 RAM 上搜索，而磁盘数据库就会慢几个数量级。
- 精确度。返回的结果列表与暴力搜索结果匹配程度如何？可以用Recall @ 10 来评估。

通常我们都会**在内存资源的限制下在速度和精准度之间权衡。**Faiss中提供了若干种方法实现数据压缩，包括PCA、Product-Quantization等向量压缩的方法 (当然还有其它的一些优化手段，但是PCA和PQ是最为核心的)，来存储十亿级别的数据。

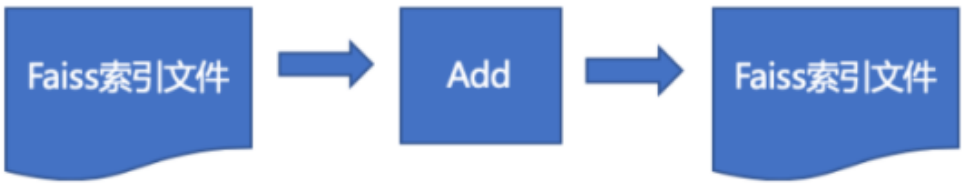
2. Faiss原理

首先来介绍一下Faiss使用时候的数据流：

全量构建索引：



增量构建索引：



在使用Faiss的时候首先需要基于原始的向量build一个索引文件，然后再对索引文件进行一个查询操作。在第一次build索引文件的时候，需要经过Train和Add两个过程；后续如果有新的向量需要被添加到索引文件，只需要一个Add操作从而实现增量build索引，但是如果增量的量级与原始索引差不多的话，整个向量空间就可能发生了一些变化，这个时候就需要重新build整个索引文件，也就是再用全部的向量来走一遍Train和Add，至于具体是怎么Train和Add的，就关系到Faiss的核心原理了。

Faiss的核心原理其实就两个部分：

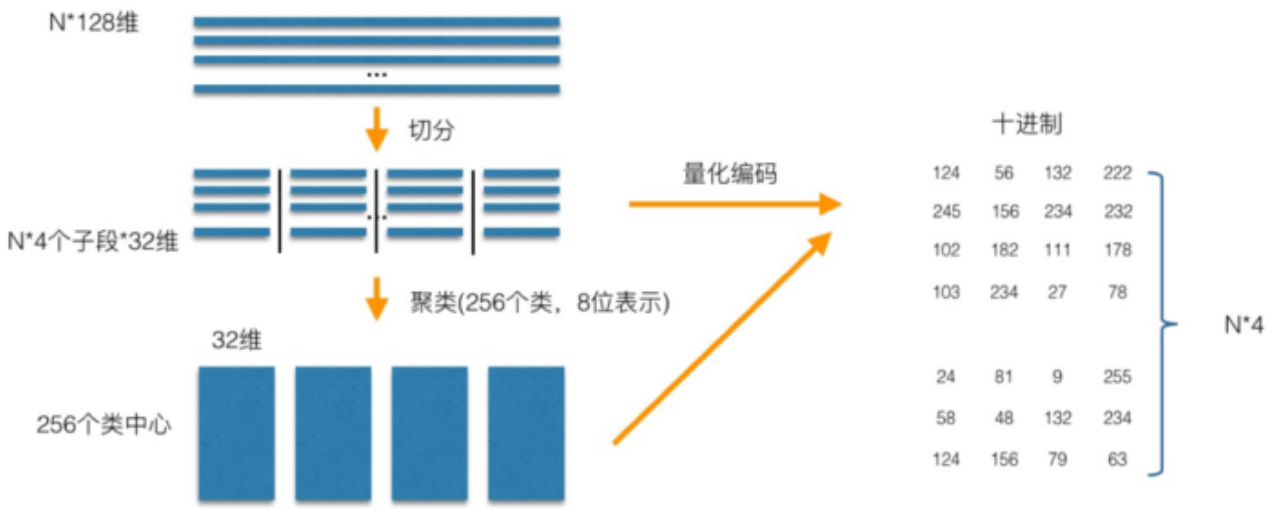
1. **Product Quantizer, 简称PQ.**
2. **Inverted File System, 简称IVF.**

2.1 Product Quantizer

矢量量化方法，即vector quantization，其具体定义为: 将向量空间的点用一个有限子集来进行编码的过程。常见的聚类算法，都是一种矢量量化方法。而在ANN(Approximate Nearest Neighbor,近似最近邻) 搜索问题中，向量量化方法又以**乘积量化**(PQ, Product Quantization)最为典型。

2.1.1 Pretrain

PQ有一个Pre-train的过程，一般分为两步操作，第一步Cluster，第二步Assign，这两步合起来就是对应到前文提到Faiss数据流的Train阶段，可以以一个128维的向量库为例：



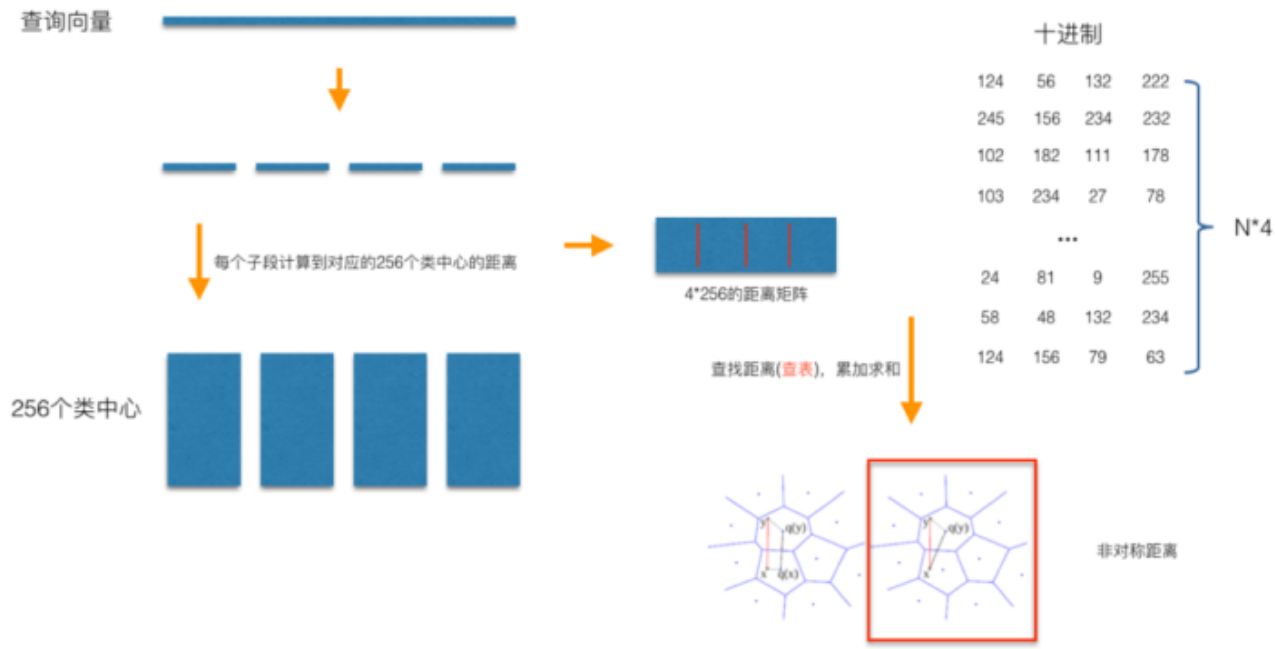
PQ乘积量化的核心思想还是**聚类**，K-Means就是PQ乘积量化子空间数目为1的特例。

在做PQ之前，首先需要指定一个参数M，这个M就是指定**向量要被切分成多少段**，在上图中M=4，所以向量库的每一个向量就被切分成了4段，然后把所有向量的第一段取出来做Clustering得到256个簇心（256是一个作者拍的经验值）；再把所有向量的第二段取出来做Clustering得到256个簇心，直至对所有向量的第N段做完Clustering，从而最终得到了256*M个簇心。

做完Cluster，就开始对所有向量做Assign操作。这里的Assign就是把原来的N维的向量映射到M个数字，以N=128，M=4为例，首先把向量切成四段，然后对于每一段向量，都可以找到对应的最近的**簇心 ID**，4段向量就对应了4个簇心 ID，一个128维的向量就变成了一个由4个ID组成的向量，这样就可以完成了Assign操作的过程 -- 现在，128维向量变成了4维，每个位置都只能取0~127，这就完成了向量的压缩。

2.1.2 查询

完成了PQ的Pre-train，就可以看看如何基于PQ做向量检索了。查询过程如下图所示：

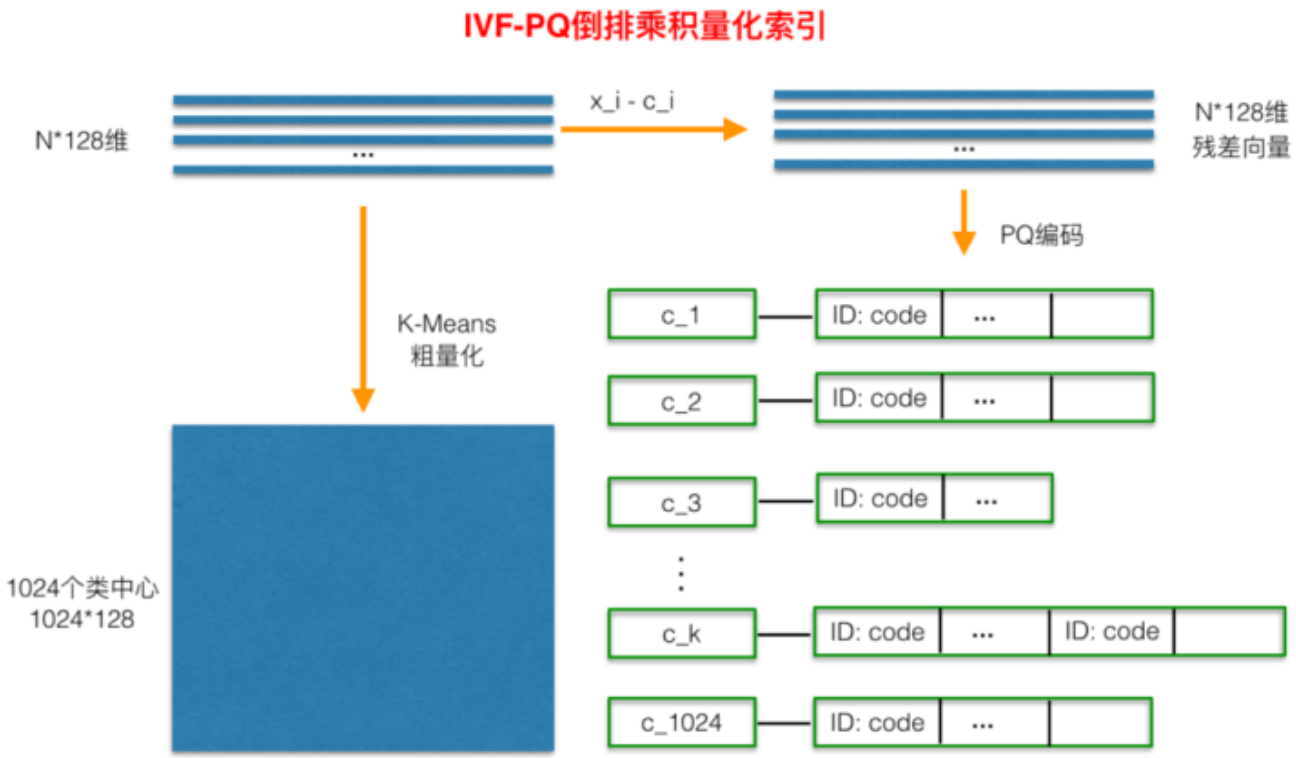


同样是以N=128，M=4为例，对于每一个查询向量，以相同的方法把128维分成4段32维向量，然后计算每一段向量与之前预训练好的簇心的距离，得到一个4*256的表。然后就可以开始计算查询向量与库里面的向量的距离。此时，库的向量已经被量化成M个簇心 ID，而查询向量的M段子向量与各自的256个簇心距离已经预计算好了，所以在计算两个向量的时候只用查M次表，比如的库里的某个向量被量化成了[124, 56, 132, 222], 那么首先查表得到查询向量第一段子向量与其ID为124的簇心的距离，然后再查表得到查询向量第二段子向量与其ID为56的簇心的距离……最后就可以得到四个距离d1、d2、d3、d4，查询向量跟库里向量的距离d = d1+d2+d3+d4。所以在提出的例子里面，使用PQ只用4×256次128/4维向量距离计算加上4xN次查表，而最原始的暴力计算则有N次128维向量距离计算，很显然随着向量个数N的增加，后者相较于前者会越来越耗时。

2.2 Inverted File System

PQ优化了向量距离计算的过程，但是假如库里面的向量特别多，依然逃不了一个遍历整个库的过程，效率依旧还是不够高，所以这时就有了Faiss用到的另外一个关键技术——Inverted File System。

倒排乘积量化(IVFPQ)是乘积量化(PQ)的更进一步加速版。其加速的本质就是在前面强调的是加速原理：brute-force搜索的方式是在**全空间**进行搜索，为了加快查找的速度，几乎所有的ANN方法都是通过对全空间分割(聚类)，将其分割成很多小的子空间，在搜索的时候，通过某种方式，**快速锁定**在某一（几）子空间，然后在该（几个）子空间里做遍历。在上一小节可以看出，PQ乘积量化计算距离的时候，距离虽然已经预先算好了，但是对于每个样本到查询样本的距离，还是得老老实实挨个去求和相加计算距离。但是，实际上我们感兴趣的是那些跟查询样本相近的样本(下面称之为"感兴趣区域")，也就是说老老实实挨个相加其实做了很多的无用功，如果能够通过某种手段快速将全局遍历锁定为感兴趣区域，则可以舍去不必要的全局计算以及排序。倒排PQ乘积量化的"倒排"，正是这样一种思想的体现，在具体实施手段上，采用的是通过聚类的方式实现感兴趣区域的快速定位，在倒排PQ乘积量化中，聚类可以说应用得淋漓尽致。



要想减少需要计算的目标向量的个数，做法就是直接对库里所有向量做KMeans Clustering，假设簇心个数为1024。那么每来一个query向量，首先计算其与1024个粗聚类簇心的距离，然后选择距离最近的top N个簇，**只计算查询向量与这几个簇底下的向量的距离，计算距离的方法就是前面说的PQ** 。Faiss具体实现有一个小细节，就是在计算查询向量和一个簇底下的向量的距离的时候，所有向量都会被转化成与簇心的残差，这应该就是类似于归一

化的操作，使得后面用PQ计算距离更准确一点。使用了IVF过后，需要计算距离的向量个数就少了几个数量级，最终向量检索就变成一个很快的操作。

3. Faiss应用

3.1 IndexFlatL2

这种索引的方式是用暴力的(brute-force)精确搜索计算L2距离。

```
import faiss                   # make faiss available
index = faiss.IndexFlatL2(d)   # build the index, d = 128, 为dimension
index.add(xb)                  # add vectors to the index, xb 为 (100000,128)大小的numpy
print(index.ntotal)            # 索引中向量的数量, 输出100000

k = 4                          # we want to see 4 nearest neighbors
D, I = index.search(xq, k)     # xq为query embedding, 大小为(10000,128)
## D为查询得到的物品embedding与query embedding的距离,大小(10000,4)
## I为和query embedding最接近的k个物品id, 大小(10000,4)
print(I[:5])                   # neighbors of the 5 first queries
```

输出：

```
100000
[[ 381  207  210  477]
 [ 526  911  142   72]
 [ 838  527 1290  425]
 [ 196  184  164  359]
 [ 526  377  120  425]]
```

IndexFlatL2的结果精确，可以用来作为其他索引测试中准确性程度的参考。当数据集比较大的时候，暴力搜索的时间复杂度很高，因此我们一般会使用其他方式的索引。

3.2 IndexIVFFlat

为了加快搜索的速度，我们可以将数据集分割为几部分，将其定义为Voronoi Cells，每个数据向量只能落在一个cell中。查询时只需要查询query向量落在cell中的数据了，降低了距离计算次数。这就是上文说的倒排索引方法。

IndexIVFFlat需要一个训练的阶段，其与另外一个索引quantizer有关，通过quantizer来判断属于哪个cell。IndexIVFFlat在搜索的时候，引入了nlist(cell的数量)与nprob(执行搜索的cell数)参数。增大nprobe可以得到与brute-force更为接近的结果，nprobe就是速度与精度的调节器。

```
import faiss                   # make faiss available
nlist = 100
k = 4

quantizer = faiss.IndexFlatL2(d)  # d = 128, dimension
index = faiss.IndexIVFFlat(quantizer, d, nlist, faiss.METRIC_L2)

index.train(xb) ## xb: (100000,128)
index.add(xb)
index.nprobe = 10              # 默认 nprobe 是1 ,可以设置的大一些试试
D, I = index.search(xq, k)
print(I[-5:])                  # 最后五次查询的结果
```

3.3 IndexIVFPQ

IndexFlatL2 和 IndexIVFFlat都要存储所有的向量数据。对于超大规模数据集来说，可能会不大现实。因此IndexIVFPQ索引可以用来**压缩向量**，具体的压缩算法就是PQ。

```
import faiss

nlist = 100
m = 8 ##每个向量分8段
k = 4 ##求4-近邻
quantizer = faiss.IndexFlatL2(d)    # 内部的索引方式依然不变
index = faiss.IndexIVFPQ(quantizer, d, nlist, m, 8) # 每个向量都被编码为8个字节大小
index.train(xb)
index.add(xb)
index.nprobe = 10
D, I = index.search(xq, k)          # 检索
print(I[-5:])
```

参考资料:

图像检索：再叙ANN Search

Faiss | 哈喽哈咯

https://github.com/facebookresearch/faiss

Faiss从入门到实战精通_bitcarmanlee的博客-CSDN博客