# 向量检索架构设计

## 1.向量检索理论

### 1.1 乘积量化

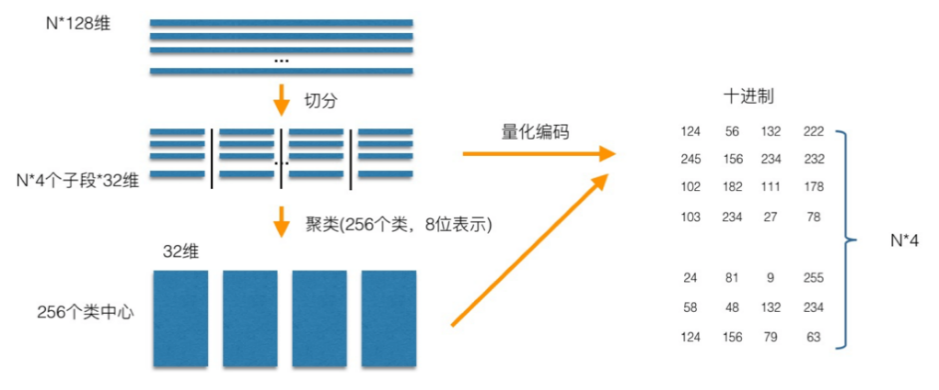
#### 1.1.1 核心思想

分段: 向量进行进行均匀分段划分

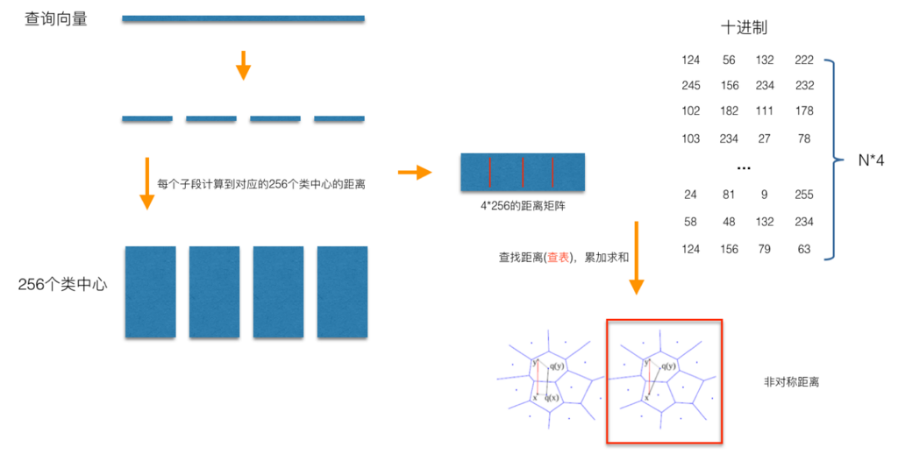
聚类: 分段进行聚类，每个类有一个编号和类中心坐标

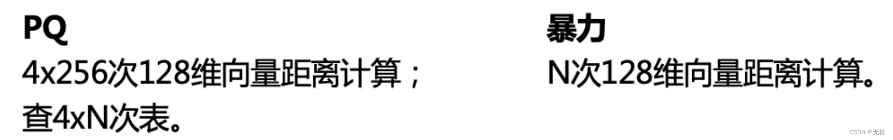
#### 1.1.2. 实现方式：

**训练阶段**： 针对 N 个训练样本，假设样本维度为 128 维，我们将其切分为 4 个子空间，则每一个子空间的维度为 32 维，然后我们在每一个子空间中，对子向量采用 KMeans 对其进行聚类（图中示意聚成 256 类），这样每一个子空间都能得到一个码本。这样训练样本的每个子段，都可以用子空间的聚类中心来近似，对应的编码即为类中心的 ID。



**查询阶段**： PQ 同样在计算查询样本与 dataset 中各个样本的距离，只不过这种距离的计算转化为间接近似的方法而获得。查询向量来到时，按训练样本生成码本的过程，将其同样分成相同的子段，然后在每个子空间中，计算子段到该子空间中所有聚类中心的距离，如图中所示，可以得到 4\*256 个距离，这里为便于后面的理解说明，可以把这些算好的距离称作距离表。在计算库中某个样本到查询向量的距离时，比如编码为 (124,56,132,222) 这个样本到查询向量的距离时，我们分别到距离表中取各个子段对应的距离即可，比如编码为 124 这个子段，在第 1 个算出的 256 个距离里面把编号为 124 的那个距离取出来就可，所有子段对应的距离取出来后，将这些子段的距离求和相加，即得到该样本到查询样本间的非对称距离。所有距离算好后，排序后即得到我们最终想要的结果。





### 1.2 基于倒排的量化乘积

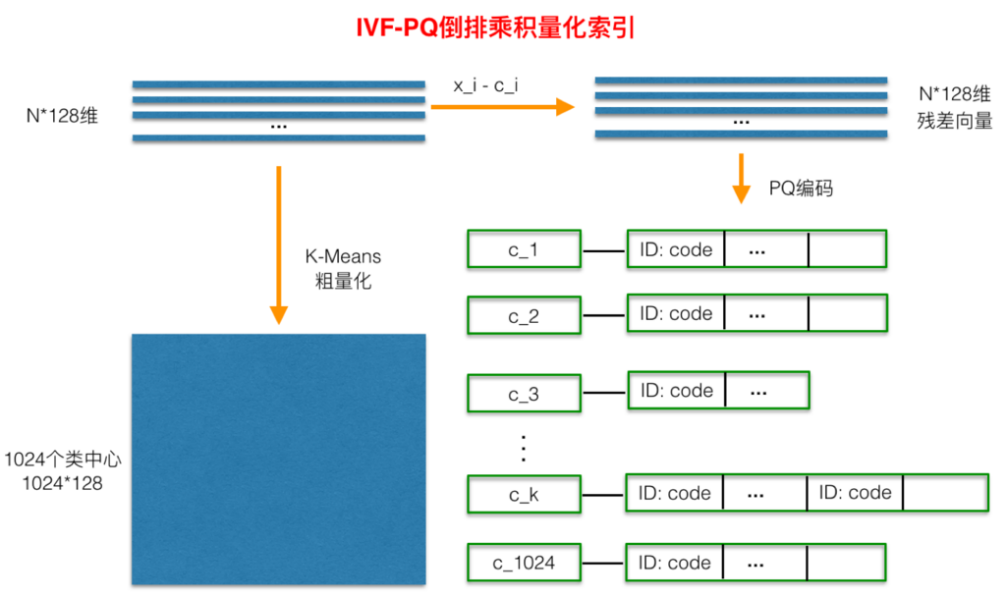
#### 1.2.1 核心思想

1. 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。 PQ 乘积量化的更进一步加速版。

2. 采用的是通过聚类的方式实现感兴趣区域的快速定位

3. IVF本身的原理比较简单粗糙，其目的是想减少需要计算距离的目标向量的个数，做法就是直接对库里所有向量做KMeans Clustering，假设簇心个数为1024，那么每来一个查询向量，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

#### 1.2.2 设计图



### 1.3 聚类算法

#### 1.3.1 Kmeans聚类算法

**1. 基本思想**

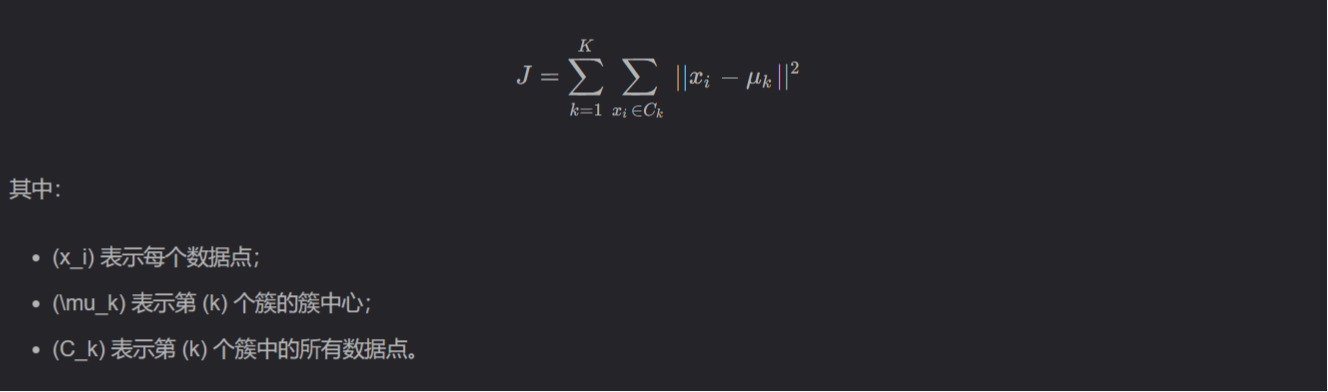
K-means 通过**迭代优化**的方式将数据划分为 (K) 个簇。每个数据点分配到距离最近的**簇中心**（*Centroid*），并不断更新簇中心的位置，直到结果收敛

**簇中心**（Centroid）：簇的中心，通常是簇内所有数据点的平均值。

**簇**（*Cluster*）：由相似数据点构成的组。

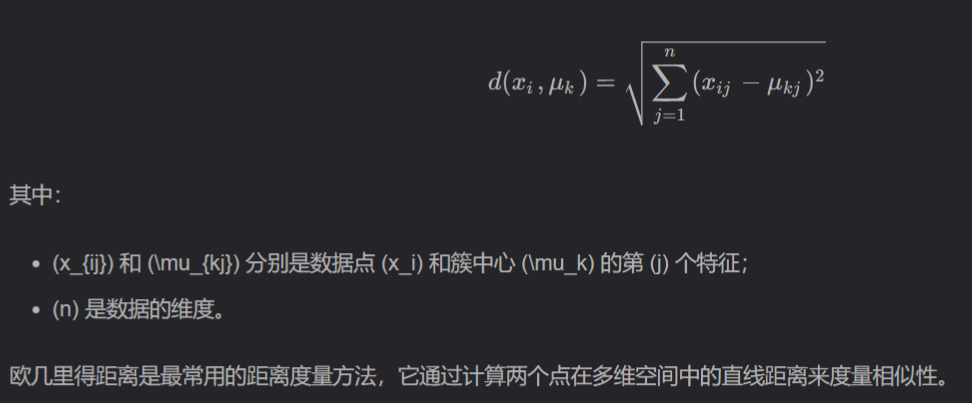
**2. 目标函数**

K-means 的目标是最小化簇内平方误差和（Sum of Squared Errors, SSE），即所有数据点到其对应簇中心的距离平方和。目标函数公式为：



**3. 距离度量**

K-means 通常使用欧几里得距离（Euclidean Distance）来计算数据点到簇中心的距离：



**4. 训练步骤**

初始化：从数据集中选择 (K) 个点作为初始簇中心。这些点将作为初始簇的中心点。

1. 随机选择1个点作为第一个簇中心
2. 从余下点中选择与第一个点最远距离的点作为第二个簇中心
3. 从余下点钟选择与已有簇中心综合距离最远的点

Weight求法看备注

1. 重复3直到获得k个簇

分配簇：对每个数据点，计算其到每个簇中心的距离，并将其分配到最近的簇。

更新簇中心：对于每个簇，重新计算簇内所有数据点的均值作为新的簇中心。

重复迭代：重复步骤 (2) 和 (3)，直到簇中心不再发生变化或达到预设的迭代次数。

**5. weight(centers,vector)**

距离列表： 计算vector与已有初始化中心的距离，得到距离列表disList，并从小到大排序

距离比值： 向量距离从前往后相除得到比值列表，divList,并从小到大排序

代码如下：

|  |
| --- |
| **private float** weight(List<Float> centerDiffs) {  **if** (centerDiffs.size() < 2) {  *//权重不需要改变* **return** 1.0f;  }   Collections.*sort*(centerDiffs);  **float** minWeight = 1.0f;  **for** (**int** i = 1; i < centerDiffs.size(); i++) {  **float** tmpWeight = centerDiffs.get(i - 1) / centerDiffs.get(i);  **if** (tmpWeight < minWeight) {  minWeight = tmpWeight;  }  }   **return** minWeight; } |

## 2.代码案例

### 2.1 训练PQ模型

|  |
| --- |
| **package** com.bird.vector;  **import** com.bird.vector.utils.ArrayTools; **import** lombok.extern.slf4j.Slf4j; **import** org.apache.commons.lang3.tuple.Pair;  **import** java.util.Comparator; **import** java.util.List;  */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@description：*** *\** ***@author：*** *liuxiangqian  \** ***@date：*** *2024/10/17  \*/* @Slf4j **public class** PQTest {  *//测试： 查询最近的n个聚类中心的(id,dis)* **private static double**[] *vector* = { 468.27878, 63.006165, 350.5396,…};   **private static float**[] *floatVector* = ArrayTools.*toFloatArr*(*vector*);   **public static void** main(String[] args) {  **int** pqSegmentCount = 16;  **int** clusterCount = 16;  **int** maxIterCount = 100;  **int** vectorDimension = 1024;   *//测试： 训练* String csvFile = **"data/test.txt"**;  EmPQ pq = **new** EmPQ(pqSegmentCount, clusterCount, maxIterCount, vectorDimension);  pq.train(csvFile);   List<List<Pair<Integer, Float>>> centersList = pq.search(*floatVector*, 10);  ***log***.info(**"聚类中心:{}"**, centersList);   *//测试： 量化得到最匹配的聚类中心id* List<Integer> clusterIds = pq.pq(*floatVector*);  ***log***.info(**"量化结果:{}"**, clusterIds);   *//测试： 模型存储与加载* String modelDir = **"pqmodel/"**;  pq.store(modelDir);  pq = **new** EmPQ(pqSegmentCount, clusterCount, maxIterCount, vectorDimension);  pq.load(modelDir);  centersList = pq.search(*floatVector*, 10);  clusterIds = pq.pq(*floatVector*);   ***log***.info(**"聚类中心:{}"**, centersList);  ***log***.info(**"量化结果:{}"**, clusterIds);  } } |

### 2.2 索引构建与存储

|  |
| --- |
| ***log***.info(**"--------------增量添加数据与检索测试----------------"**); *//构建向量索引类，并进行向量注入* EmIndex emIndex = **new** EmIndex(pq); String csvFile = **"data/test.txt"**; Pair<List<Integer>, List<**float**[]>> idsAndVectorsPair = VectorTools.*laodIdsAndVectors*(csvFile); List<**float**[]> vectors = idsAndVectorsPair.getValue(); List<Integer> ids = idsAndVectorsPair.getKey(); emIndex.addVector(vectors, ids); ***log***.info(**"总向量数:{} "**, vectors.size());  ***log***.info(**"---------------内存损耗------------------"**); *//打印内存损耗 memoryDetect*();  *//向量索引存储与加载* ***log***.info(**"---------------存储索引-------------------------"**); emIndex.store(indexDir, indexPrefix); |

### 2.3 索引加载与向量检索

|  |
| --- |
| EmIndex emIndex = **new** EmIndex(pq); emIndex.load(indexDir, indexPrefix); List<Pair<Integer, Float>> docs = emIndex.searchDocs(*floatVector*, clusterTopn, topn); **for** (**int** i = 0; i < docs.size(); i++) {  Pair<Integer, Float> pair = docs.get(i);  **int** id = pair.getKey();  **float** dis = pair.getValue();   List<**float**[]> vectors = emIndex.getIndexVectors().get(id);  StringBuilder sb = **new** StringBuilder();  **for** (**int** j = 0; j < vectors.size(); j++) {  **float**[] vector = vectors.get(j);  String embedding = Arrays.*toString*(vector);  sb.append(embedding).append(Separators.***COMMA***);  }  ***log***.info(**"id:{} dis:{} embedding:{}"**, id, dis, sb.toString()); } |