南京大学

研究生毕业论文
(申请硕士学位)

论文题目 近似最近邻搜索算法研究与应用

作者姓名 刘凤山

学科、专业名称 软件工程

研究方向 人工智能

指导教师 申富饶 教授

2021 年 5 月 30 日

学　　号：MG1833098

论文答辩日期：2021 年 5 月 24 日

指 导 教 师：　　**(签字)**

Approximate Nearest Neighbor Search Algorithms and their application

by

Fengshan Liu

Supervised by

Professor Furao Shen

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
MASTER
in
Software Engineering

Department of Computer Science and Technology
Nanjing University

May 28, 2021

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：近似最近邻搜索算法研究与应用

软件工程 专业 2018 级硕士生姓名：刘凤山

指导教师(姓名、职称)：申富饶 教授

摘　要

在大数据时代，每天都有海量的数据产生，以深度学习为代表的技术被广泛应用于从复杂的数据中提取信息。深度学习技术通过将复杂的对象如文本、语音、图像等编码为高维向量来表示对象，并通过向量之间的距离来度量对象之间的相似性。在构建各种应用系统如搜索推荐系统时，常常需要在大规模、高维度的向量上进行相似特征检索。为了解决这类大规模高维向量上的相似特征检索问题，各种近似最近邻搜索算法被提了出来。

尽管存在各种各样的近似最近邻搜索算法，实际应用中的相似向量检索仍然存在不少困难。首先是面对种类繁多的近似最近邻搜索算法，如何针对具体应用场景选取适合的应用算法。其次对于召回率要求极高场景，现有很多算法都是不适合的。最后是对于大规模数据进行相似特征检索的时候，算法构建的索引模型会占用大量内存，如何减少构建的索引的大小。本文将上述问题总结为通用场景下近似最近邻搜索算法的选择问题，以及内存资源受限的情况下如何有效降低索引模型大小的问题。对于前者，HNSW 算法是目前的主要选择，但是仍然存在问题。

针对上述问题，本文做了以下工作：

(1) 对于 HNSW 现有的节点删除方法存在的节点大量删除之后部分搜索请求返回结果数量不足的问题，本文提出了新的节点删除算法 HNSW Mutual-Remove，成功且高效地解决了该问题。此外，本文的实验结果也表明基于 GPU 加速的线性扫描算法也拥有接近 HNSW 的性能，非常适合于对召回率要求极高的场景。

(2) 对于近似最近邻搜索算法所构建的索引模型内存占用大的问题，本文基于 HNSW 对其进行了深入分析。本文认为可以分别采用对高维向量进行压缩以

及采用更加轻量级的组织数据的数据结构来解决该问题。IVF-HNSW 算法虽然结合了以上两点对索引大小进行了极大比例的压缩，但是其构建速度慢。对此本文提出了新的索引构建方法，我们称之为 Balanced IVF-HNSW。该方法在大幅加快索引的构建速度的同时仍能够保证较高的召回率。

(3) 成功将上述优化后的算法应用到微信大规模分布式近似最近邻搜索组件 SimSvr 中，该组件能够完成对数十亿级规模的数据的高效索引与检索，已经广泛应用于微信搜一搜、看一看等业务中。

本文的实验结果以及相关算法在 SimSvr 中的应用经验表明，本文提出的 HNSW Mutual-Remove 成功解决了 HNSW 缺少适合的节点删除算法的问题，有效提升了 HNSW 的稳定性，同时本文提出的 Balanced IVF-HNSW 有效加快了 IVF-HNSW 构建索引的速度，让 IVF-HNSW 在内存资源受限场景下变得更加实用。

关键词： 信息检索；近似最近邻搜索；方法选择；模型压缩

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Approximate Nearest Neighbor Search Algorithms and their application

SPECIALIZATION: Software Engineering

POSTGRADUATE: Fengshan Liu

MENTOR: Professor Furao Shen

ABSTRACT

In the era of big data, massive amounts of data are generated every day, and technologies represented by deep learning are widely used to extract information from complex data. Deep learning technology expresses objects by encoding complex objects such as text, speech, images, etc. into high-dimensional vectors, and measures the similarity between objects through the distance between the vectors. When building various application systems, such as recommendation systems, it is often necessary to perform similar feature retrieval on large-scale, high-dimensional vectors. To solve similar feature retrieval problems on large-scale high-dimensional vectors, various approximate nearest neighbor search(ANNS) algorithms have been proposed.

Although there are various approximate nearest neighbor search algorithms, applying ANNS in practical applications still has many difficulties. Firstly, it is difficult to choose suitable ANNS algorithms for specific application scenarios between a variety of ANNS algorithms. Secondly, for scenarios that require extremely high recall rates, many existing algorithms are not suitable. Finally, when performing similar feature retrieval on large-scale data, the index model constructed by the algorithm will take up a lot of memory. How to reduce the size of the constructed index. This paper summarizes the above problems as the selection problem of ANNS algorithm in general scenarios and the effective reduction of index size under limited memory resources conditions.

In response to the above problems, this article has done the following work:

(1) Our analysis shows that HNSW is the preferred algorithm in general scenarios.

But the deletion of nodes in HNSW index may cause insufficient results returned by some search requests. After analyzing the graph structure of HNSW index, we find out that this is caused by the lack of a correct node deletion algorithm in HNSW. Based on this observation, we propose a new node deletion algorithm, which we call HNSW Mutual-Remove, that effectively avoids the problem of insufficient search results while deleting nodes efficiently. Besides, through the analysis and experiment of HNSW and linear scan, we show that in scenarios where a very high is required, GPU-based acceleration linear scan can be considered

(2) We analyze the problem of high memory usage of ANNS algorithms by studying the index structure of HNSW. We think that this can be solved by compressing the high-dimensional vectors or adopting more lightweight ANNS algorithms. Although the IVF-HNSW algorithm combines the above two points to significantly reduce the index size, its index construction speed is slow. So we propose a new index construction method, which we call Balanced IVF-HNSW, that could greatly accelerate the construction of the index while maintaining good performance.

(3) Successfully apply the above-optimized algorithm to wexin's distributed large-scale data ANNS library SimSvr, which has indexing billions of data from many applications inside wexin.

The experiments and application of our optimized ANNS algorithms to SimSvr show that the HNSW Mutual-Remove proposed by us successfully solves the problem that HNSW lacks a suitable node deletion algorithm and makes HNSW more versatile. At the same time, the Balanced IVF-HNSW proposed in this paper effectively accelerates the indexing speed of IVF-HNSW, making IVF-HNSW more practical in scenarios with limited memory resources.

KEYWORDS: Information Retrieval, Approximate Nearest Neighbor Search, Method Selection, Model Compression

目　　录

插图清单

附表清单

第一章　绪论

1.1　研究背景与意义

在大数据时代，每天都有海量的数据产生，以深度学习[1] 为代表的技术被广泛应用于从复杂的数据中提取信息。深度学习技术通过将复杂的对象如文本、语音、图像等编码为高维向量来表示对象，并通过向量之间的距离来度量对象之间的相似性。在构建各种搜索推荐系统时，常常需要在大规模、高维度的向量上进行相似特征检索。由于维度灾难的问题[2]，对于高维向量的相似特征检索并不存在比线性扫描更好的方法。然而，在实际应用通常不严格要求检索最相似的向量，而是允许一定的误差。基于这样的考虑，K 近似最近邻搜索被提出来了（K Approximate Nearest Neighbor Search，以下简称 K-ANNS）。K-ANNS 算法将需要索引的向量通过特殊数据结构（索引）进行组织，能够在速度大幅优于线性扫描的情况下返回 K 个近似最近邻。一般用召回率衡量 K-ANNS 算法的搜索质量，定义为返回的 K 个近邻中真实的最近邻的数量与 K 的比值。目前，在大规模高维向量上进行相似特征检索时也面临诸多挑战，如索引构建耗时长，索引的动态更新、以及如何平衡资源占用与搜索效率与质量之间的关系等问题。因此，面对越来越多的数据，研究构建速度快、可扩展性高、资源占用少、搜索效率高的 K-ANNS 算法成为重要课题。

近似最近邻搜索算法被应用于各种涉及大规模、高维度的相似向量检索问题的领域中，如数据库、数据挖掘、机器学习、计算机视觉等领域。下面列举近似最近邻搜索的常见应用。

图片搜索。移动互联网的兴起让每个人都成为了数据的生产者和消费者，各种和数据相关的需求也就应运而生。图片搜索就是给定一张图片，搜索与其相似的图片，比如对于某中植物的图片，我们可以通过互联网搜索相似的图片，进而判断该植物的种类。目前，各大搜索引擎都支持该功能。得益于深度学习的发展，在图片搜索系统中，一般采用神经网络[1] 将图片编码为高维向量，这样既有利于减少内存占用，也有利于比较图片之间的相似度。编码后的图片会用近

似最近邻搜索算法组织成特定数据结构放在内存中，由于图片搜索系统需要索引的图片数量多，在加上编码后的图片本身维度就很高，因此整个索引的内存占用会很高。这是图片搜索系统面临的重要挑战之一。此外，所有的原始图片也需要存储下来，方便和请求匹配时进行展示。

人脸识别系统。人脸识别技术是一种重要的生物体特征识别方式。由于其只要部署好，就可以采用几乎不需要被识别个体参与的方式识别个体，已被广泛应用于安防等领域。一个人脸识别系统包含一系列技术，如图像采集、人脸定位、人脸预处理与识别、身份确认等技术。通常，在人脸数量比较少的时候是不需要采用近似最近邻搜索的，比如在学校，通常需要识别的用户数量级在几千到数万之间，这种情况下采用近似最近邻搜索在速度上并不比线性扫描更有优势。由于近似最近邻搜索并不能保证返回最近邻，因此最终需要综合采用其他信息来进行人身份的确认。

推荐系统。推荐系统技术是多种技术的集合体。在各大互联网公司的产品中都有广泛的应用，它会根据用户的行为，如点击、浏览等，来给用户建模、然后推荐它认为用户喜欢的内容。比如电商场景中，应用会根据用户过往浏览记录和购买记录来给用户推荐它认为用户会喜欢的商品，网易云音乐会根据用户过往听歌记录给用户推荐歌曲等等。在这些场景中，用户的浏览记录、听歌记录以及推荐内容等被编码为向量、然后通过相似向量检索给用户推荐适合的内容。这些系统通常面临着大量的数据以及高维的编码向量，同时推荐系统也天然容忍一定的误差，这种场景非常适合近似最近邻搜索。

在小规模的数据上，最近邻搜索问题可以使用暴力便利的方式解决。但是在大规模数据集上这个方法会耗费大量时间，很难满足应用需求。对于低维数据，存在一些方法，如 k-d 树[3] 能够解决最近邻搜索问题。但是随着维度升高，由于维度灾难问题的存在，k-d 树的搜索速度退化为了线性扫描。这里维度灾难问题表现为，在高维数据上，一个点与其他点的最大距离和最小距离之间的差距可以任意小，使得 k-d 树中请求数据所在的子区域几乎和树的其他所有子区域都存在重合，使得搜索效率几乎等价于线性扫描。因此近似最近邻搜索问题面临的第一大挑战就是探索在召回率上能够更加接近线性扫描，而搜索效率有远优于线性扫描的算法。

近似最近邻搜索算法通过特定的数据结构将数据索引起来，为了加快搜索

速度，近似最近邻搜索通常都会采取某种策略先缩小需要搜索的数据的范围，然后在小范围内进行搜索，比如倒排索引[4] 会先将数据进行聚类根据聚类中心将数据分桶，搜索过程中先确定一批候选的桶，然后在候选桶中进行搜索。在近似最近邻搜索算法中，召回率和召回时间是不可两全的，更高的召回率往往需要更长的搜索时间，比如倒排索引中，搜索更多的通能够提高召回率，但是也降低了搜索速度。

为了保证召回速度，通常会将近似最近邻搜索算法构建的索引放在内存中以加快访存速度。但是对于大规模、高维度的数据，这会带来大量的内存开销。通常可以采用矢量量化（Vector Quantization 或 Product Quantization, 一般简称 PQ）[5] 对向量进行压缩、使用内存占用更少的近似最近邻搜索算法以及借助其他廉价存储设备如硬盘来分担内存开销等方法来降低内存开销。但是这些方法无疑都会带来算法性能及效果的下降。如何尽量减少内存占用并保持良好的性能是一个重要问题。

近似最近邻算法通常是计算密集的算法，使用各种软件硬件方法对计算进行加速是重要的课题。通常在大规模数据上构建索引会比较耗时，如果构建模型的算法支持并行化那么就可以大大加快构建模型的速度。此外由于近似最近邻搜索需要把大量的计算花费在向量之间的距离计算上，借助硬件对向量之间的计算进行加速也是必不可少的，比如基于 SIMD[6] 技术或者基于通用图形计算单元（Graphics Processing Unit，简称 GPU）[7] 进行加速。

由于近似最近邻搜索算法的数量不胜枚举，如何针对一个特定问题选取适合的近似最近邻搜索算法也成为了一个重要的问题。本文会结合自己的实践经验，对针对具体场合如何选取适合的搜索算法给出一些通用准则。

1.2 研究现状

K-ANNS 是信息检索领域的常用方法，是 K 最近邻搜索（K Nearest Neighbor Search, 下文称 K-NNS）的近似解。K-NNS 定义如下：给定一个空间 M 中点的集合 S 与请求 q，和定义在该空间中的距离函数 $d(x,y) \quad x,y \in M$, 返回与 q 距离最近（最相似）的 K 个点。K-ANNS 允许返回的 K 个点存在一定的错误。并用召回率来衡量搜索结果的质量，假定集合 S 中离 q 最近的 K 个点集合的为 G，

近似最近邻算法返回的 K 个近似最近邻的集合为 P，则召回率定义为 $\frac{|P \cap G|}{K}$。

在实际应用中，空间 M 通常是 d 维的向量空间，距离函数通常是欧式距离或余弦距离。

衡量一个 K-ANNS 算法通常会考虑一下几个指标，召回率、每条数据平均召回时间、索引构建时间、索引大小、是否支持增量式构建索引这几个因素。在实际应用中，会综合考虑这些因素选取合适的算法。

根据算法基本思想的不同，可以将近似最近领搜搜算法分为四大类别。分别是基于局部敏感哈希（locality-sensitive hashing，后文简称 LSH）[8] 的方法、基于空间划分的方法、基于图的方法、基于对向量进行压缩的方法。

基于局部敏感哈希的方法本质上使用了一系列满足特殊数学性质的哈希函数。哈希函数本质上是将空间中的一系列点映射到另一个空间中，而局部敏感哈希函数是一种尽可能保证距离相近的点被映射到同一个位置的函数。这样在检索的时候，就可以使用 LSH 函数将请求映射到相近的位置，然后进行检索。由于单个 LSH 函数不能保证能将请求映射到它所有邻近的点，所以实际应用中需要结合多个局部敏感哈希函数来提高召回率。当然这样产生的代价是更长的搜索时间以及更多的计算资源占用。局部敏感哈希的代表算法有，SRS[9]，QALSH[10]。

基于空间划分的方法的采用了分而治之的算法设计思想。它们会试图对待索引的数据进行各种形式的拆分，将大数据集的问题转化为小数据集的问题。k-d 树就是一种分而治之的方法，通过采用树形数据结构组织数据从而将数据集进行拆分。基于 k-d 树，随机 k-d 树[11] 也被提出来了，它在选取划分数据的平面时引入了随机性，并且构建索引的时候会构建多份随机 k-d 树索引。另一种流行的划分方法是使用聚类方法对数据进行划分，分层 k-means 树[12] 采用递归的方法不断对数据进行聚类，每一次都将数据根据聚类中心划分成 K 份，然后对被划分出的数据重复此操作。在实践中，通过对数据进行聚类划分，一般只划分一次，然后构建倒排索引。这种方法可以和基于图的方法结合加快搜索速度，或者与基于量化的方法对向量进行压缩。

基于图的方法在召回率和召回时间上的表现是目前的 state-of-the-art。基于图的方法的核心思想是将被索引的向量当做图中的一个节点。通过在节点之间添加连边将节点直接或者间接连接起来。这样将最近邻的搜索过程转化为在图上进行贪心搜索的过程。Hierarchical navigable small world graph（下文称 HNSW）

[13] 是最流行的基于图的方法、它采用了类似于跳表[14] 的数据结构。不同的是跳表的每一层是“一维”的，而 HNSW 的每一层是" 二维" 的。这种分层结构有利于加快搜索效率。HNSW 构建的图结构中每一层采用了一种用来近似 k-nearest neighbor graph（简称 K-NNG）的结构，这种结构尽可能保证任意两点之间存在一条通路，提高检索的召回率。HNSW 在为每个节点添加邻居时，借鉴了 sparse neighborhood graph（下文称 SNG）[15] 的构图方法，这种方法尽可能保证每个节点的邻居分布在其周围各个方向上，加快搜索过程的收敛。另外还有一些构图核心思想和 HNSW 类似的方法还包括 NSG[16] 和 Vamana[17]，它们在性能上相对于 HNSW 并没有明显提升，反而 HNSW 的增量式构建方法让其在实践中更受欢迎。

基于向量压缩的方法是采用矢量量化等方法对向量进行压缩表示的方法。量化本质上是对空间中的一系列点通过聚类的方法进行划分，然后每个点用其聚类中心进行表示。通过将聚类中心进行编码，每个数据在内存中只需要存储其聚类中心对应的编码，从而达到压缩的效果。通常量化方法可以和前面提到的各种构建索引的方法进行结合使用。虽然量化会导致距离计算精度的损失，但由于其强大的压缩能力，其在实践中有广泛的应用。

1.3 研究内容

本论文结合实际应用需求，从两方面展开了研究。第一个是通用场景下的近似最近邻算法的选取问题。本文认为 HNSW 作为最流行算法的 ANNS 算法之一，是大多数场景的首要选择，但是对于召回率要求极高（比如严格要求为 100%）的场景，可以采用基于 GPU 加速的普通线性扫描算法。同时对于 HNSW 目前的部分情况下召回结果不足的问题，本文提出了新的算法解决了该问题。第二个是对于内存资源受限的场景，如何减小近似最近邻算法所构建的索引的大小。本文对该问题进行了深入的分析与总结，并将其中的代表算法 IVF-HNSW[18] 进行了优化，让其更加实用。除此之外，本文还讨论了如何构建工业级的大规模数据的近似最近邻检索组件，并成功将上述对近似最近邻搜索算法的优化应用于微信分布式近似最近邻搜索组件 SimSvr [①] 中，取得了很好的效果。

[①] https://mp.weixin.qq.com/s/rXXm6c8LrTqqP4iWf9mtxA

本文主要研究内容可以概括如下：

对 HNSW 的优化。现有的 HNSW 算法存在大量删除节点导致部分搜索请求返回结果数量不足的情况，本文通过分析 HNSW 的构图过程以及最终的图结构得到该问题是现有的算法缺少节点删除算法导致的。由于 HNSW 构建的图结构是单向图，被删除节点并不知道有哪些节点拥有指向它的边，这让节点删除变得比较困难，也就导致了上述的问题。为了解决上述问题，本文提出了新的节点删除算法，我们称其为 HNSW Mutual-Remove。HNSW Mutual-Remove 算法并没有引入额外的存储开销，而是将被删除节点作为搜索目标，通过在图中进行搜索来获得有哪些点将被删除节点作为邻居，从而尽可能多地获得与被删除节点有关的信息，从而达到删除节点的目的。由于 HNSW 搜索速度很快，节点删除也能很快完成。我们的实验显示 HNSW Mutual-Remove 算法成功解决了由于节点删除导致搜索请求返回结果不足的问题，验证了该算法的有效性。

HNSW 与线性扫描的对比分析。通过对 HNSW 与线性扫描的分析与实验，本文认为虽然 HNSW 是现在综合表现最好的算法之一，但是基于 GPU 进行加速的线性扫描在中等规模的数据集上同样拥有不俗表现。在对召回率要去极高的场景下，可以考虑采用基于 GPU 加速的线性扫描算法。

对于近似最近邻搜索算法存在的索引内存占用大的问题，本文基于 HNSW 算法对此进行了深入分析。本文认为可以分别从对高维向量进行压缩以及采用更加轻量级的组织数据的结构来解决这个问题。IVF-HNSW 算法虽然结合了以上两点对索引大小进行了极大比例的压缩，但是其构建速度慢。本文针对这个问题提出了新的构建方法，我们称之为 Balanced IVF-HNSW。该方法在大幅加快索引的构建速度的同时尽量保证较高的召回率，使得内存占用极少的 IVF-HNSW 真正成为实用的方法。

在上面工作的基础上。本文分析了如何构建工业级的大规模数据的近似最近邻检索组件，并将 HNSW 算法（包含新提出的 HNSW Mutal-Remove）和 Balanced IVF-HNSW 应用到了微信大规模分布式相似特征检索组件 SimSvr 中。该组件能够完成对数十亿级规模的数据的高效索引与检索，已经广泛应用于微信搜一搜、看一看等业务中。

此外，针对目前最流行算法 HNSW 的官方实现 hnswlib [①]在实现中存在的一

[①]https://github.com/nmslib/hnswlib

些问题，我对其进行了修复并已经被官方采纳。

1.4 论文纲要

本文主要研究对常用近似最近邻搜索算法的优化与应用。主要包括提出新的节点删除算法解决 HNSW 现有节点删除算法存在的问题，对 IVF-HNSW 构建索引速度慢的问题进行了优化。此外也给出了将上诉两种算法和量化算法进行结合的实践经验。全文共分为六章，第一章为绪论，主要介绍了近似最近邻搜索问题研究的背景和意义，以及本文的研究内容；第二章介绍了近似最近邻搜索算法近年来的进展；第三章介绍了 HNSW 算法现有节点删除算法存在的问题与解决方案，以及对线性扫描的分析与实验，还介绍了如何用 HNSW 算法对聚类算法进行优化；第四章介绍了近似最近邻搜索算法存在的内存占用过高的问题并提出了一些缓解措施，针对其中常用方法 IVF-HNSW 训练过程慢的问题，提出了解决方法；第五章介绍了对上述算法的实际应用，即微信大规模分布式相似特征检索系统 SimSvr。这一章着重介绍了 SimSvr 的设计准则，系统组成以及关键技术；第六章是全文的总结，以及对未来工作的一些展望。

第二章　相关工作

2.1　近似最近邻搜索

最近邻搜索（Nearest Neighbor Search，简称 NNS）定义为：给定一个在某度量空间 X 中的 n 个点的集合 $S = \{s_1, s_2, \ldots, s_n\}$，对集合 S 进行处理，使得对于请求 $q \in X$, 能够高效地找到 x 的最近邻。当然这个问题很容易扩展为 K 最近邻搜索。对于低维情况，这个问题已经能够被很好地解决[19]。但是对于高维的情况，受维度灾难的困扰，人们在这个问题上一直止步不前。这个问题最早在 1960s 就被提出来了[20]。

因此人们也转向了对近似最近邻搜索的研究，也就是允许返回离请求 $1+\epsilon$ 倍最近邻距离的点（ϵ-Apprxoimate Neareast Neighbor Search, 简称 ϵ-ANNS）。LSH 就是 ϵ-ANNS 的代表算法，它是由 Pioter Indyk 和 Rajeev Motwani 在 1990s 提出的[8]。基于 LSH 有许多衍生算法，如 SRS，QALSH 等。这些算法上虽然数学上有很好的理论下界，但是还是被后来的基于图的算法所超越。

基于图的算法如 HNSW 等会占用大量内存，可以使用矢量量化技术来有效减少内存占用。IVFADC[21] 是一种将空间划分，矢量量化进行结合的方法，非常适合于对上亿级别的数据数据进行索引。IVF-HNSW 是在索引 IVFADC 的聚类中心时使用 HNSW 算法，进一步加快检索速度。

其实基于向量量化的方法和基于局部敏感哈希的方法有许多类似之处。但是实践中前者常用于作为内存压缩的一种手段和其他方法结合使用，而后者一般是单独作为一个方法，所以这里将二者分开列举。下面针对每个类别的算法分别进行介绍。

2.2　基于局部敏感哈希的方法

基于局部敏感哈希的方法通过选定一组适当的哈希函数，每次从中随机选择一个 LSH 函数对数据进行处理。这类方法在理论上保证了最坏情况下的搜索

结果质量、效率以及索引大小。

2.2.1 局部敏感哈希函数

LSH 函数族 F[22] 定义在在度量空间 M，概率 p_1, p_2 和距离函数 $d(x_1, x_2)$ 上。该函数族里面的任意函数 $h : M \to S$ 将度量空间 M 中的元素映射到桶 $s \in S$ 中。对于度量空间中的任意两点 $x_1, x_2 \in M$，从 LSH 函数族 F 中采用随机均匀采样一个函数 $h \in F$, 该函数满足：

$$\begin{cases} P(h(x_1) == h(x_2)) \geq p1 & \text{, if } d(x_1, x_2) \leq d_1 \\ P(h(x_1) == h(x_2)) \leq p2 & \text{, if } d(x_1, x_2) \geq d_2 \end{cases}, \tag{2-1}$$

这里的 f 即为一个局部敏感哈希函数，这里的 (p_1, p_2, d_1, d_2) 为预先指定的概率和距离，这里的 $d(x_1, x_2)$ 表示 x_1 和 x_2 之间的距离（相似度）。对于满足上述性质的函数族 F，我们称其为 (p_1, p_2, d_1, d_2) 敏感的。

从上面的分析可以看到，任意两点之间的距离越大，它们被映射到相同的桶的概率就越小，反之就越大。我们可以把这种映射理解为一个空间划分。被划分到同一个空间的点就位于相同的桶中。显然，这样会带来一个明显的问题，位于划分边界附近的点虽然彼此距离很近，但还是有很大概率彼此被分开。这也是选择一组 LSH 函数的原因，只需要经过仔细选择 LSH 函数，就能尽最大可能保证不同的 LSH 函数不会有相同的分类边界。这样就能尽可能提高搜索时候的召回率。

因此，设计性能良好的 LSH 函数变成了重要的问题。有许多方法被提出来，比如 AND-then-OR[23] 方法、OR-then-AND[24] 方法、基于树的方法[9] 等。下面列举几个经典方法。

2.2.2 SRS

SRS 是提出来解决 ϵ-ANNS 问题的。SRS 算法将高维空间中的点映射到低维空间（通常小于 10）然后进行 K-ANNS 搜索。给定一个请求 q, 对于原空间中的任意一个点 s, 它们在原空间中的距离定义记为 d_1 ，在新空间中的距离记为 d_2。那么二者的比值满足卡方分布 $\chi^2(m)$。这里的 m 是投影后空间的维度。

SRS 的搜索过程可以分为两步：（1）在投影的空间上进行 K-ANNS。获得 $k = T'$ 个候选者；(2) 按距离（相似度）从近到远的顺序逐个检查每个候选者，直到遇上终止条件并返回目前最近的节点。这里的终止条件包括存在一个点以超过预先定义的概率阈值是请求的 ϵ 近邻或者已经访问完了全部的 T' 个候选者。

SRS 算法保证了当 $m = O(1)$ 的时候，以确定概率返回距离不大于最近邻距离 ϵ 倍的节点。算法的时空复杂度与数据条数 n 呈线性关系，而与数据维度 d 无关。

2.2.3 QALSH

传统 LSH 函数对数据的预处理过程可以表示为 $h_{\vec{a},b}(o) = \lfloor \frac{\vec{a}\cdot\vec{o}+b}{w} \rfloor$。原始向量 $\vec{o}$ 被经过一次变换之后被添加上一个随机偏移。最后除以 w 并向下取整就是将投影后的空间从原点开始分成一个个长度为 w 的小区间，也就是一个桶。这样对数据进行分桶的过程并没有请求数据的参与，因此被称为与与请求无关的数据预处理方法。这种方法有一个问题就是，对于经过投影变换并添加随机偏移的向量，分桶的边界固定在 w 的整数倍处，对于经过同样变换后落在这种边界附近的请求，它的最近邻们就横跨了左右两个桶，这样仅仅搜搜请求被映射到的桶就就获得最近邻的可能性就更小了。

为了缓解上述问题。QALSH 提出与请求相关的算法，让请求数据参与数据的预处理过程。QALSH 提出让请求数据参与对数据的分桶过程，这样使得请求数据的最近邻以更大可能性与请求数据位于同一个桶，从而提高最近邻的命中率。QALSH 先使用哈希函数对数据进行投影变换，$h_{\vec{a}}(o) = \vec{a} \cdot \vec{o}$。对于数据的分桶如下，使用请求 q 或者其经过同样哈希函数映射后的值 $h_{\vec{a}}(q)$ 作为分桶区间的中心（这里使用后者），获得一个桶的区间如下 $[h_{\vec{a}}(q) - \frac{w}{2}, h_{\vec{a}}(q) + \frac{w}{2}]$。这里的分桶方式刚好让请求成为分桶区间的中心，由于请求数据参与了最终的分桶过程，所以叫做与请求相关的数据预处理方法。这样带来的好处是显而易见的，即增大了命中最近邻的概率，同时哈希函数中也不在需要随机的偏移量 b。在数据无关的数据预处理方法中，由于分桶方法都是使用固定的 w 的整数倍作为分桶边界，因此需要增加一些随机性让在桶边界附近的点有落在另一边的可能性增加。所以数据有关的处理与处理方法省掉了这一项，有利于减少运算量。QALSH 使用了 B^+ 树索引被哈希后的函数值，然后在具体分桶的时候只需要从

树里面检索落在区间 $[h_{\vec{a}}(q) - \frac{w}{2}, h_{\vec{a}}(q) + \frac{w}{2}]$ 上的数据，并没有很大的计算开销。此外可以灵活选取参数 w 来控制检索的范围。综合上述分析，QALSH 提出了一种新颖的与请求相关的数据预处理方法，这种方法在提升最近邻的命中率的同时带来的额外的计算开销也是可接受的。

QALSH 方法针对 ϵ-ANNS 问题在前代方法上有较大提升。SRS，QALSH 这些早期的 LSH 方法并没有从数据的分布中学习哈希函数，后面的研究从数据分布中学习哈希函数，这些方法表现更好。

2.2.4 AGH

哈希函数对数据的映射过程（分桶）过程也可以理解为对数据进行二进制编码。当然，二进制编码的长度越大，哈希碰撞的概率就越小，搜索召回率也就越大。但是同时计算量和内存占用也就越高。通常基于数据无关的局部敏感哈希函数的方法需要更长的二进制编码来保持较好表现。而数据相关的局部敏感哈希方法只需要更短的二进制编码。这也使得采用数据相关的方法学习到的哈希函数通常拥有更高的召回率和更低的内存占用。在 AGH 提出之前，学习数据相关的哈希函数通常需要掌握全局的距离度量信息。而 AGH 基于图由算法自动获得该信息。AGH 是基于 Weiss 等人的方法[25] 进行改进的。Weiss 的方法主要包含三个步骤：（1）使用 n 个点构建近邻图，时间复杂度为 $O(dn^2)$；（2）计算近邻图的离散拉普拉斯矩阵的的 r 个特征向量，时间复杂度为 $O(rn)$；（3）将 r 个特征向量扩展到任意没见过的数据 $O(rn)$。这个方法存在的问题是对于非常大的 n，每个步骤都很耗时，尤其是步骤（1）。为了缓解这个问题，AGH 提出了一种无监督的哈希方法。该方法中，采用 Anchor Graph[26] 来构建近邻图。在这种图结构里面，节点之前的相似度是通过一定数量的 anchor 来度量的，通常 anchor 的数量比较少，通常为数百。通过这种方式，整个近邻图的构建过程耗时为 $O(n)$，而且可以通过增加 anchor 的数量来提高构图质量。由于 Anchor Graph 的邻接矩阵具有低秩特性，因此可以很快地计算近邻图的离散拉普拉斯矩阵的 r 个特征向量。此外，由于图结构对于未见到的点能够很好的进行编码也使得其在实践中很受欢迎。在图结构中，相近的节点往往具有相似的语义标签，尤其是在流行数据结构中，AGH 采用图型数据结构也使得它能够保留语义相近的邻居关系。

AGH 是从数据中学习与数据相关的 LSH 函数的代表。其他类似代表方法还有 Scalable Graph Hashing（简称 SGH）[27]，Neighborhood APProximation index（简称 NAPP）[28] 等。

2.3 基于空间划分的方法

近似最近邻搜索面临的问题第一是数据量大，第二是数据维度高。通过采用分而治之的策略就是将数据变 “小”，变得容易处理是一种很符合直觉的方法。而通过对数据的数量进行拆分，我们得到了数量更小的同类型问题，但是对维度的分而治之却得不到同类型的子问题。对数据集进行递归的拆分很自然就可以得到某种树形结构。这种分而治之的策略本质上是将搜索过程锁定在一个比较小的范围，这样就避免了遍历所有数据。所以分而治之本质上是对空间的划分，使得划分后空间中相似的点位于同一个或者邻近的区域。这种类似树结构的划分方式有利于在搜索的时候快速锁定目标区域，大幅加快搜索速度。基于空间划分的方式面临着和基于 LSH 的方法类似的问题，位于划分边界附近的点被划分边界分隔，导致要提高召回率就需要搜索更多的区域，从而降低搜搜速度。

下面就一些典型方法进行介绍。

2.3.1 随机 k-d 树

在 k-d 树的搜索过程中，必须采用回溯的方法来进一步搜索最近邻。当数据的维度增大，回溯过程总所需要访问的节点越来越多。以至搜索效率逐渐退化为线性扫描。随机 k-d 树通过在不同的树上独立并行进行搜索来解决这个问题。

树的构建是通过递归的选取划分平面将数据一分为二，传统方法是每次选取一个坐标轴，然后选取垂直于该坐标轴并通过该坐标轴上数据中位数的平面将数据进行拆分。在选取坐标轴的时候，每个坐标轴是轮着来的[3]，也就是说对于三维数据情况，前面选取了 x 轴，那么本次可以选择 y 轴或者 z 轴，当本轮次 x、y、z 轴都被选取完毕之后，则开始新一轮的选取。随机 k-d 树开始构建之前会对数据进行预处理，即采用 PCA 降维方法将数据映射到低维空间并使得数据的 momnent axes[11] 与数据的坐标轴是平行的。

随机 k-d 树的构建过程也和传统 k-d 树不一样。首先是在划分坐标轴的选取上，随机 k-d 树在这里引入了随机的方法。具体的，每次从当前采样方差最大的五个维度中随机选取一个维度，这里的采样方差是针对单个维度而言的，为了减少计算两使用了采样的方法。这样的好处一是引入了随机性，可以用来构建完全不同的多颗 k-d 树。此外，倾向于选取数据分布方差大的维度是因为这样有利于对数据进行更加均衡地进行拆分。选取完成划分维度之后寻取垂直于该划分维度的平面时也与传统方法不一样，随机 k-d 树选取了该维度所有数据平均数对应的垂直平面对数据进行划分。这样也有利于保证数据拆分的均衡性。

随机 k-d 树通过在树的构建过程中引入随机性使得我们可以同时构建多颗不一样的 k-d 树。由于传统方法构建 k-d 树时几乎没有随机性，所以不能通过构建多棵随机树来提高搜索效率。随机 k-d 树通过构建多颗树作为索引，在搜索的时候并行搜索多棵树。虽然多棵树的搜索过程是并行的，但是它们共享相同的结果队列（采用了优先级队列）和候选者集合。每次都选取离目标最近的点进行搜索并将当前结果中最远的点作为搜索下界。通过共享搜索结果队列和候选者集合，虽然在实践中会带来进程/线程同步的开销，但是可以避免在某些离目标很远的树节点上浪费大量时间，让搜索过程更快地朝着最优方向前进。

此外，由于随机 k-d 树是用来解决 ANNS 问题的。一般会设置一个搜索节点数量的上限，保证搜索过程在最坏情况（没有达到其他提前终止条件）下也能顺利结束。

2.3.2 分层 k-means 树

前面提到的方法无论是经典的 k-d 树还是随机 k-d 树都是采用平面将数据一分为二。但是采用平面对数据进行拆分并不一定是一个好方法，尤其在高维数据上，在某个被拆分的维度上相距较远的点可能实际距离很近。因此也产生基于其他假设的数据拆分方式。基于距离的远近来进行数据拆分是一种很符合直觉的方法。搜索的时候需要快速确定目标所在的大致区域，基于距离远近的数据拆分方式能够尽力保证与目标临近的点都在确定的搜索范围内。

基于聚类方法对数据拆分就属于上面讨论的基于距离远近进行数据划分的常用方法。分层 k-means 树使用 k-means 聚类方法 [29-30] 递归地对数据进行 K 聚类，每次聚类结束将数据拆分成 K 份，直到数据的个数不足 K 个为止。这样最

终的索引就是一个树形结构，树的每个叶子节点都包含最终不可继续拆分的点。在搜索过程中，会先从树的根节点开始搜索，每个选择最近的字节点进行搜索直至搜索万叶子节点为止。这样就形成了一条从树的根节点到叶子节点的路径。如果仅仅只搜索单个叶子节点，显然不会有很高的召回率。为了提高召回率，需要通过该路径上的节点进行回溯，搜索“附近”的叶子节点。显然这里又回到了对搜索效果和搜索效率进行权衡的时候了。回溯越多，就能便利更多的点，理论上召回率也会更高，但是也带来了更长的搜索时间。如果对路径上所有节点对应的子树进行回溯，那就退化为线性扫描了。

2.3.3 倒排索引

倒排索引本来是进行文本检索的常用数据结构，常用于搜索引擎中的大规模网页文本检索。倒排索引本质上也是一种对数据进行拆分的方法，类似于哈希将数据进行分桶一样。检索的时候先确定数据在哪个“桶”里面，然后在“桶”里面去检索数据，这也是倒排索引概念的由来。在 ANNS 问题上，已经有不少应用采用倒排索引，并将其于其他方法结合使用。如，IVFADC 将倒排索引与矢量量化进行结合[21], IVF-HNSW 将倒排索引与 HNSW 以及矢量量化进行结合[18]。在 ANNS 应用场景中，一般采用的方式是先通过聚类方式将数据进行依次拆分，构建倒排索引。IVFADC 主要是对每个数据进行压缩，即将高维数据进行压缩以减少索引内存占用。IVF-HNSW 针对千万级及其以上场景中，“桶”的数量较多，通过采用搜索速度快且准的 HNSW 来索引“桶”对应的聚类中心来加快定位待搜索的“桶”，进而加快检索速度。

2.4 基于图的方法

前面介绍了基于局部 LSH 的方法，基于空间划分的方法。这两类方法都有一个共性，就是会对数据空间进行划分。很显然基于空间划分的方法会对数据空间进行划分，但是基于 LSH 的方法呢。每个 LSH 函数事实上都会对数据空间进行拆分，这里因为局部敏感哈希就如其名字一样，具有局部性。如果这种函数不在数据空间中存在着数据拆分的边界，那就不存在什么局部性了，极端情况所有数据都被映射到同一个桶（这里桶的意思是具有相同哈希函数值的点的

集合），虽然保证了每个点都与其近邻都处于同一个桶，但是由于所有点都在一个桶，其实是没有意义的。这种显示的空间划分方式必然导致位于边界附近的点与其近邻被分隔开，破坏了他们之间的邻居关系。事实上这种邻居关系是非常有用的，它指示了点之间的“亲疏”关系，对于最近邻搜索具有重要的指导意义。

基于图的方法关注到了可以利用每个点同其附近点的信息来指导搜索过程。在基于图的方法中，每个数据也就是高维空间中的一个点被作为图中的一个节点，每个点与其临近的点之间用边连接，如果两个点之间有边连接，我门称这两个点建立了邻居关系。对于图中的每个节点，它的邻居距离目标的距离信息就能够指导搜索过程，如果某个节点的邻居远离搜索目标，那就表示我门不需要探索该邻居了，此时转而探索那些离搜索目标更近的邻居能够让我们更快地接近搜索目标。

可以通过一个实例来展示该过程。如图 2-1 所示，我们将城市作为图中的节点，图中的边表示两个城市之间 存在某种交通方式将两个城市连接起来，比

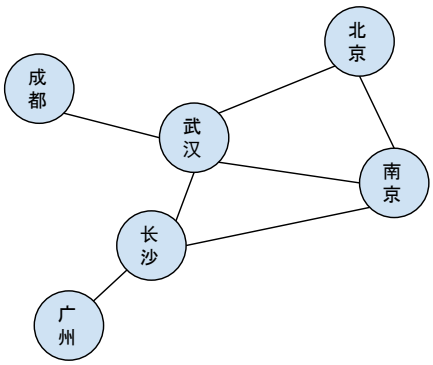


图 2-1: 在图上寻路实例。图中的节点表示一个城市，图中的边表示城市之间存在某种互通的交通方式。当某人位于武汉且目的地是广州的时候，由于他发现通往北京的交通方式明显离目的地更远，因此他的下一站的选择是长沙。

如飞机。现在你从成都出发前往广州，当你来到武汉市的时候，你有不少选择，但是很显然你会选择直接去长沙，接着去广州。这里虽然在武汉的时候，你可以选择去北京然后去南京、长沙、接着去广州。但是很显然你知道去北京会绕远路，所以你选择下一站是长沙。这就是图里面的寻路过程，它能让你根据一些局部信息指导你更快接近终点。

基于图的方法的核心是如何构图。一般来说，所构建的图需要在以下两点上尽量做得完美。第一，构建的图中，任意两点之间需要存在一条距离递减的

路径。第二，这条路径的长度尽可能地短。第一点保证了搜索过程能够找到最近邻，第二点是为了确保搜索时间不会太长，在极端情况下所有节点位于一条链表上，平均路径长度就达到 O(n) 了。目前还没有算法能够高效构建满足以上两个条件的图结构。在图理论中，德劳内三角拆分构建的图能够满足条件一[31]，而 K 近邻图（K-Nearest Neighbor Graph，简称 K-NNG）是德劳内三角剖分的子图[32]，一般基于图的算法都会构建近似 K-NNG。小世界网络[33] 是一种满足图中任意节点时间的路径长度都处于 $log(n)$ 级别的图，是一种性质上符合上面要求二的图结构，但是其构建复杂度太高，很少有方法会直接构建这种图结构。

基于图的方法是目前的 state-of-the-art，下面介绍目前最流行的基于图的方法。

2.4.1 HNSW

HNSW 算法是目前综合表现最好的近似最近邻搜索算法之一，也是最具代表性的基于图结构的近似最近邻搜索算法。ANN-Benchmarks [34] 是目前评测各大近似最近邻算法实现库最权威的平台①，HNSW 算法的官方实现 hnswlib，其在 ANN-Benchmarks 上各个任务上表现都非常不错。图 2-2是 ANN-Benchmarks 上各大近似最近邻搜索算法在 ANN_SIFT1M [21] 数据集上的表现。

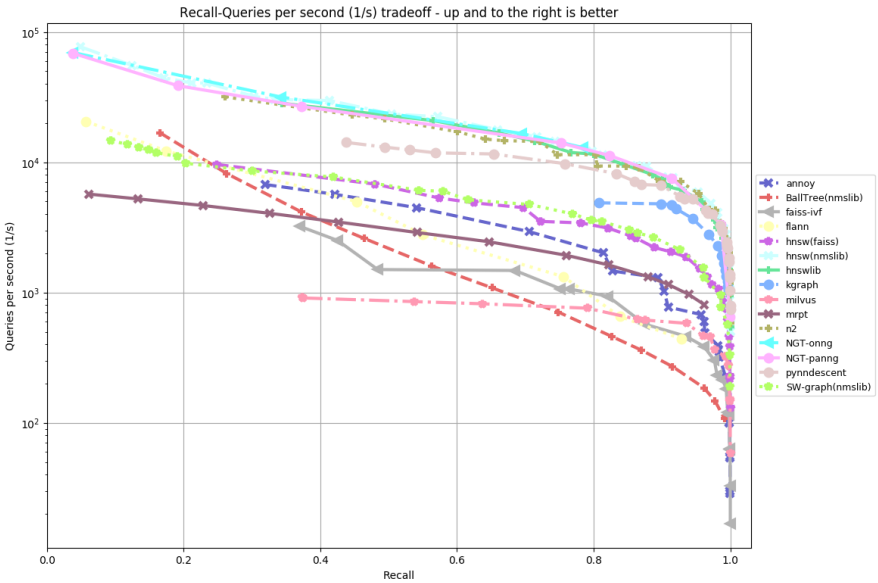


图 2-2: ANN-Benchmarks 上各大近似最近邻搜索算法在 ANN_SIFT1M 数据集上的表现。横坐标表示召回率，纵坐标表示每秒完成搜索请求数。

①https://github.com/erikbern/ann-benchmarks

HNSW 算法是在 NSW[35] 的基础上改进而来的。基于图的算法的核心要素之一是要保持局部的近邻信息，以保证在各个方向上都具有很好的探索性。K-NNG 图由于是德劳内图的子图，能够尽力保证两点之间具有距离递减的路径。所以 NSW 在构图的时候也在力图构建近似 K-NNG 图。NSW 采用一种增量式的构图方式，对于每个新数据，会在已经构建完成的图上面搜索离该待插入数据最近的 K 个点作为其邻居。所以这里构建的是近似 K-NNG。一方面，在图上进行的贪心搜索过程并不能保证能够搜索到该图上离新数据最近的 K 个点。另一方面，由于目前图上的数据并不包含全部数据，也就无法满足得到全局的最近的 K 个邻居。综合上面两个点，NSW 构建的图是近似 K-NNG 的图。近似 K-NNG 能够保证图在搜索的过程中具有较好的探索性，不至于陷入局部最优，最终提高搜索的召回率。但是这样的图并不能保证这样在这样的图上的搜索距离足够短。

NSW 的构图过程通过构建近似小世界网络的图来缩短图上搜索路径的长度。而这个过程和构建近似 K-NNG 图的过程并不冲突。由于新的数据节点是依次加入图中的，我们就会发现，在构图的初期，由于图上节点相对较少，两个相距较远的点会有机会成为邻居。这里的距离远近是可以理解为和整个图的尺度是匹配的。在最终构图完成后，在构图前期许多节点之间建立的边就成了沟通距离很远的节点的 “高速公路”。这样借助于这些高速公路就能大幅缩短跨越整个图进行搜索的路径长度。正是构图初期建立的跨越整个图的 “高速公路” 承担起了高速公路的作用，缩短了任意节点间的平均路径长度，让最终构建的图除了能够近似 K-NNG 图，还能拥有近似小世界网络的能力。

基于以上两点，NSW 一经提出就取得了非常好的效果。虽然 NSW 具有很好的效果，但是它仍然存在不少问题。比如搜索过程容易陷入局部最优，在高维数据上表现不理想，此外搜索算法的搜索复杂度是 $O(log(n)^2)$ 级别，还没有达到 $O(log(n))$ 级别。为了改进 NSW 存在的种种问题，HNSW 被提出来了。

HNSW 借助了跳表[36] 这种数据结构的基本思想。跳表是一种分层结构，每一层是一个链表，由于每个节点能够获得的最大层次越大概率越低，导致大部分节点盘踞在较低层次的链表中，这种结构使得跳表在搜索的时候具有 $O(log(n))$ 的复杂度。HNSW 在构图的时候，把每一层的链表 “替换” 为 NSW 这种 “二维” 结构，这种结构让 HNSW 也获得了 $O(log(n))$ 的搜索复杂度。HNSW 除了具有

分层结构外，在邻居选取上也进行了优化，进一步降低的了搜索过程陷入局部最优的可能性。HNSW 目前是在各种数据集或者实际任务中综合表现最好的算法之一。其官方实现 hnswlib 也被业界广泛采用。

2.4.2 其他基于图的算法

这里介绍两个在 HNSW 提出后出现的两个基于图的方法，NSG 和 Vamana。它们的基本构图思想和 HNSW 大同小异，但是并没有在性能上相对于 HNSW 有明显优势，反而它们的构图过程都不是增量式的，并不太适合实际应用场景。

NSG 在构图的之前会先预处理所有数据，得到一个全局的近似 K-NNG。这个过程不仅比较耗费内存和计算资源，而且需要较长的时间来构图。而 Vamana 在构图之前会使用所有的数据构建随机图。这两个算法的初始化方法使得它们并不适合增量式地插入新数据。在邻居选取上，三个算法都采用了类似的策略，Vamana 在这里引入了额外的参数来控制图的直径。Vamana 这种做法会进一步减少在一些类似长条形状数据上构建的图的直径，这样能够缩小搜索路径长度。这样虽然不会带来搜索效率的有效提升，但是由于 Vamana 构建的图被保存到了 SSD 上面，缩小路径长度有利于较少访问 SSD 的次数，加快搜索速度。

2.5 基于向量压缩的方法

矢量量化是用于信号处理里面常用的方法，可以用来对数据进行有损压缩。由于近似最近邻搜索通常处理的都是高维数据，而且构建的索引（包含所有高维数据）也是放在内存里面的。通常原始高维数据贡献了构建的索引的大部分内存开销，所以对于内存紧张的场景，急需要对高维数据进行压缩。自 IVFADC 将矢量量化和倒排索引结合以来，矢量量化已经成为了一种通用的技巧，可以和各种近似最近邻搜索方法结合使用。faiss[37] 就将矢量量化作为了一种通用技巧，可以与其他集成的近似最近邻搜索算法结合使用。

何恺明提出了 OPQ[38] 对 PQ 进行优化，进一步降低了矢量量化前后的点与点之间的距离损失。矢量量化是一中将原空间拆分为多个子空间的笛卡尔积的方法，OPQ 提出的技巧能够进一步降低这种拆分过程量化误差。目前 OPQ 已经成为矢量量化中的常用技巧。

除了矢量量化，还存在标量量化[39]（Scalar Quantization，简称 SQ）也是对数据进行压缩的常用方法。高维向量的每一个数据通常是浮点数，浮点数在计算机系统中占用 32 个比特，使用标量量化能够将单个浮点数占用的比特数压缩到四倍及其以上，同时保持较低的量化误差。

这里介绍了矢量量化和标量量化，他们是通用的数据压缩方法。它们已经成为了一种通用技巧，可以同各种近似最近邻搜索算法结合，在大幅压缩构建的索引大小的同时尽最大可能保留点与点之间的距离。

2.6 本章小结

本章介绍了近似最近领搜索的相关工作。我把近似最近领搜索的相关方法分为了基于局部敏感哈希的方法、基于空间划分方法、基于邻接图的方法、以及基于矢量量化的方法。基于局部敏感哈希的方法虽然是数学上保证了最坏情况下的搜索速度与搜索质量。但是它的召回率和召回时间上比不上基于图的方法。基于空间划分的方法是一类基于分而治之的思想的方法，这些方法非常符合人的思想直觉。其中倒排索引额外开销少，非常适合大规模数据集（千万级及其以上）。基于图的方法是在召回率和召回时间上表现最好的方法，这使得它们成为实践中选取方法的首选。基于向量压缩的方法通过将通用的矢量量化/标量量化技巧和以上各种近似最近邻搜索算法结合，在保持较高的召回率的同时大幅压缩索引的大小，节省内存资源。

如上所示，近似最近邻搜索算法种类繁多，如何根据实际情况进行灵活选取，以及各种算法在应用中会存在什么问题，这些问题怎么解决。都会在接下来几章进行介绍。

第三章　对 HNSW 算法的分析与优化

第二章介绍了现存的各种近似最近邻搜索算法，同时指出就在召回率和召回时间上的表现来看，HNSW 算法是目前最好的算法之一。但是 HNSW 算法存在由删除节点导致的搜索返回结果数量不足的问题，本文分析这是由于现有 HNSW 算法缺少节点删除算法导致的。针对这个问题，本文提出了新的 HNSW 节点删除算法 HNSW Mutual-Remove，成功解决了该问题，并通过实验验证了算法 HNSW Mutual-Remove 的效果。随后本章通过实验充分对比了 HNSW 算法与线性扫描算法。验证了基于 GPU 加速的线性扫描算法拥有接近 HNSW 的速度，非常适合于对召回率要求极高的场景。最后本章也通过实验验证了 HNSW 对聚类算法 k-means 的加速效果。

3.1　通用场景下的 ANNS 算法选取

在评判某个算法的时候，我们首先会考虑该算法是否能够达到预期的效果。对于近似最近邻搜索问题，我们首要关注的是算法是否能够在优于线性扫描（后文的表格和图中也将其称为 Linear Scan）的时间里返回 K 个最近邻。这里涉及的主要指标是召回率和召回时间。我们所说的通用场景，就是主要关注这两个指标的场景。虽然在 ANN Benchmarks 上面已经指出 HNSW 算法是目前在这两个指标上面表现最好的方法之一。但是被很多人忽视的是，有的时候线性扫描也是可选的方法之一。甚至在对召回率要求极高的场景，线性扫描成为了唯一选择。此外，HNSW 算法本身存在当索引中的数据被大量删除之后，部分请求返回的结果数量不足 K 的问题，这一问题一定程度上影响了它的通用性。

上文提到线性扫描现在成为了一个可选项。这是由于现在随着计算机硬件技术的不断发展，计算机的算力在不断增强，经过合理优化之后的线性扫描程序就能在较短时间内完成运算，此外如果用户对召回时间并没有严格要求，使用线性扫描也是合理的。线性扫描还有另外一个优势就是它不需要对数据进行费时的预处理以及复杂繁琐的模型调参。

在通用场景下，HNSW 算法与结合硬件进行优化后的线性扫描是首要选择。

本章后续将介绍对 HNSW 存在问题的改进以及 HNSW 与线性扫描的对比实验。

3.2 对 HNSW 算法的改进

HNSW 算法已经被广泛应用到各种实际应用中。本节根据实际应用场景分析了 HNSW 算法目前存在的不足，即不能很好地支持节点的修改和删除，随后本节提出了新的节点修改与删除算法，最后给出该新提出算法的复杂度分析以及在实验数据集上的效果。

3.2.1 HNSW 的搜索算法

HNSW 算法的核心之一是它的搜索过程。它是一个带有回溯的贪心搜索过程。虽然 HNSW 构建的图是一个多层的结构，但是在每层的搜索过程中使用的都是相同的搜索算法，只是根据不同需要，参数 ef 的取值不同。在构建索引的时候，ef 的取值为 $efConstruction$。当索引构建完成之后，搜索最底层的图的时候 ef 取值为 $efSearch$，而在搜索高层网络（第一层之上的图网络）的时候，ef 的取值一般为 1。设置为 1 是因为在高层进行搜索的时候并不需要太过精确的结果，而是只需要在尽量靠近目标节点的同时耗费尽可能少的计算量。一般模型构建的时候采用的参数 $efConstruction$ 取值较大，这是为了保证构图质量，提高搜索时候的召回率。

HNSW 算法每一层的搜索过程如算法 3.1所示。算法搜索的入口点 ep 是上一层中离请求 q 最近的节点，对于最高层，搜索入口点为预先定义好的节点（如最先添加的最大层数最大的节点）。在实践中，可能会构建各种各样的索引，以应对各种各样的需求。比如，有的时候用于构建索引的节点数可能会很少，有的时候搜索的召回数量 K 可能会很大，有的时候需要删除或者更改某个节点。我们需要算法在任何时候都能正常运行，即具有很高的稳定性。

HNSW 的搜索算法是 HNSW 构图的核心步骤，也是下文对 HNSW 进行深入分析的基础。

3.2.2 HNSW 算法存在的问题

现有的 HNSW 算法还存在下列问题：

算法 3.1 HNSW 每一层图上的搜索算法[13]

```
1: function HNSW-SEARCH-LAYER(l, ep, ef, q)
输入:
    l 表示搜索网络的层次
    ep 表示搜索的入口节点
    ef 搜索过程中回溯队列的最大长度
    q 请求数据
输出:
    ef 个离 q 最近的节点
2: Vis ← ep //已经访问过的节点的集合
3: Candi ← ep //候选者集合，即待访问的节点的集合
4: Ans ← ep //保存离请求数据 q 最近的点的优先级队列
5: while |Candi| ≠ 0 do
6:    cur ← Candi 中离 q 最近的节点
7:    Candi ← Candi − cur
8:    tmp ← Ans 中离 q 最远的节点
9:    if distance(tmp, q) ≤ distance(cur, q) then
10:      break
11:   end if
12:   for neb ∈ neighborhood(cur) do
13:      if neb ∉ Vis then
14:         Vis ← Vis ∪ neb
15:         if |Ans| < ef  or  distance(neb, q) < distance(tmp, q) then
16:            Candi ← Canid ∪ neb
17:            Ans ← Ans ∪ neb
18:            if Ans > ef then
19:               移出优先级队列 Ans 中离 q 最远的节点
20:            end if
21:         end if
22:      end if
23:   end for
24: end while
25: return Ans
```

不支持节点删除操作。在图中删除一个节点除了删除节点本身以外，还需要删除与该节点相关联的边。本质上 HNSW 算法构建的图结构是一个有向图，每个节点维护一个邻居集合，即从该节点出发的边的终点集合。在这种图结构中，被删除节点只知道以自己为起点的边，而不知晓有哪些节点包含以被删除节点为终点的边。图 3-1解释了这一现象。

不支持节点修改操作。图中的每个节点是空间中的高维向量。对于每个索引，这些向量都有其特殊意义。比如，针对图片检索任务，这些向量就是图片经过深度神经网络编码后的高维向量，即图片的特征向量。针对其他任务，这

些向量可以是语音、文本等内容经过编码后得到的特征向量。随着时间的迁移，某些被编码的事物可能发生了改变，比如推荐系统中会动态调整用户对各种物品的兴趣，那么相应的特征向量也需要做调整。如果这种调整只涉及索引中很少部分的数据，那么从头构建索引是一项非常费时费力的工作，我们迫切希望能够直接在现有索引上直接修改对应节点的向量。但是 HNSW 算法是不支持直接修改节点的高维向量，即节点的坐标信息的。这是因为，HNSW 所构建的图的邻居关系是由节点间的位置关系决定的。直接修改坐标而不修改邻居关系会破坏正常的图结构。因此，需要考虑新的方法修改节点的坐标信息。

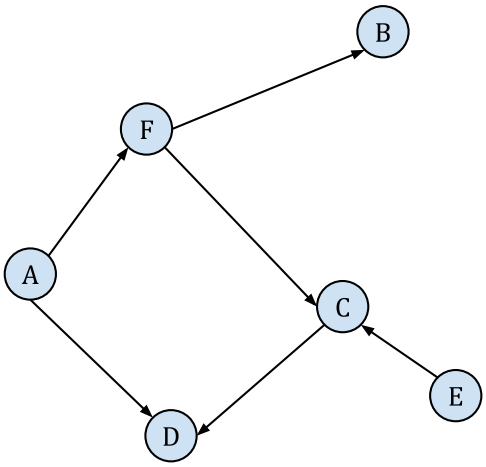


图 3-1: 单向图。在这种图中每个节点只知道从自己出发的边的信息。比如节点 F 知道节点 B 与 C 是它的邻居，而不知道节点 A 也将其作为了邻居。

由于节点修改过程可以通过删除原节点然后插入修改过的节点来完成，因此以上两个不足之处本质上都是由于 HNSW 不支持节点删除导致的。由于 HNSW 算法并不支持节点删除操作。为了应多实践中对节点删除操作的需求，hnswlib 的做法是将被删除节点标记为删除。算法 3.1在搜索过程中遇到这些节点时，会直接忽略它们，即既不将他们加入候选者集合 $Candi$ 也不访问它的邻居。下面讨论这种方式会产生的问题。

返回结果不足 K 个。对于每一个请求，给定需要返回结果的数量 K，在索引中节点数量 n 满足 $n \geq K$ 的时候，都应当保证返回结果的数量是 K 个。这里的 n 不包含被删除的节点。但是，当部分节点被删除之后，就可能出现返回结果数量不足 K 个的情况。下面先回顾 HNSW 处理搜索请求的整个过程。

索引构建完成之后的搜索过程如下，对于每个搜索请求，从最顶层开始逐层向下搜索，直至算法 3.1 在最底层结束。在最底层进行搜索的时候，参数 ef

取值为 $efSearch$，在其他层次进行搜索的时候，ef 取值为 1。这里，$efSearch$ 的大小至少为 K。更大的 $efSearch$ 能够获得更高的召回率，但是也会带来搜索速度的下降，所以 $efSearch$ 需要根据实际需求灵活选取。从这里可以看出，决定搜索返回结果数量的是在最底层的搜索过程。下面，讨论出现返回结果不足 K 个的具体情形。

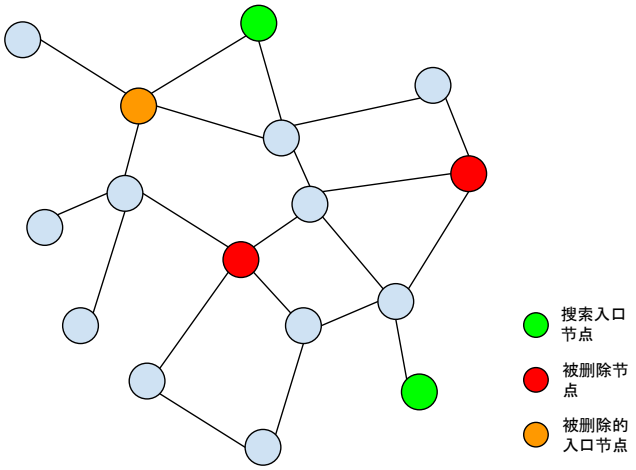


图 3-2: 搜索入口被标记删除的示例。这里边的方向被略去。绿色节点表示搜索入口节点，红色节点表示被删除节点，橙色节点表示被删除的搜索入口节点。通过增加搜索入口节点的数量，能够增加搜索的时候有可用的搜索入口节点的概率。

通常情况下，当搜索入口节点附近的节点被删除之后，就很可能导致搜索提前终止，从而导致返回结果不足 K 个。极端情况下，如果搜索入口 ep 直接被删除的话，会导致整个搜索过程被迫终止。图 3-2展示了为了保证搜索的时候有可用的搜索入口节点所采取的措施，但是该技巧还是无法保证算法在最坏的情况下能够稳定运行。此外，该技巧针对搜索最开始的时候，即从最顶层开始的时候。在其他层进行搜索的时候，搜索入口点都是上一层搜索过程中遇见的离请求最近的节点。一般情况下，导致返回结果不足 K 个的情况发生于在最底层进行搜索的时候，这种时候如果搜索入口点附近的节点被删除，就很有可能出现搜索结果不足 K 的情况。如图 3-3所示就是这种情形。由于绿色节点 F 的邻居都被删除了，搜索过程提前终止，只返回一个结果，即 F。当然，如果离 F 较远的节点被大量删除，也会导致提前终止的情况，但是返回的结果的数量会更加接近 K。因此，对于较大的 K，更有可能出现返回结果数量不足 K 的问题。

这里讨论了由于节点被标记为删除导致的节点数量不足 K 的问题。我们发现，直接将节点标记为删除虽然简单直接。但是极端情况下会出现出人意料的

结果。如果需要定期删除大量节点，这种方法是不适合的。

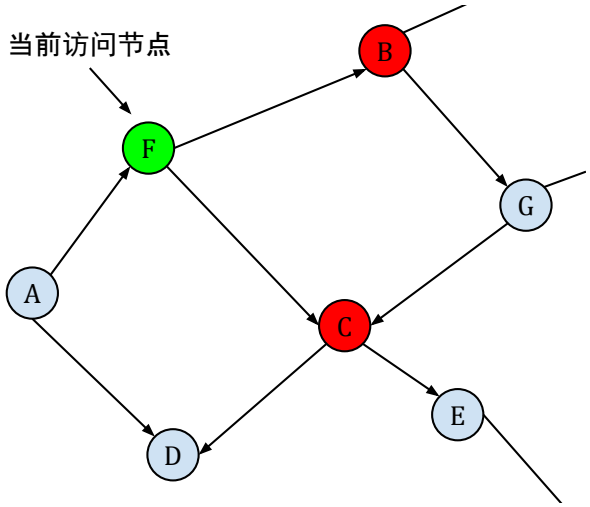


图 3-3: 搜索返回结果不足 K 个。图中绿色节点表示当前访问节点，红色节点表示被删除节点。由于当前访问节点的“邻居”都被删除，导致算法 3.1直接返回，导致返回结果不足 K 个。

返回结果命中率低。前面讨论了由于直接将被删除节点标记为删除而导致的召回结果不足的问题。由于该问题是很容易直接观测的，比如我们通过统计召回结果的数量就很容易发现召回结果不足的问题。但是，这种删除节点的方式也会导致某些很不容易被观测到的问题。这就是本节将要讨论的就是这样的问题：由于部分节点被标记为删除，影响到了图的连通性，导致在响应部分搜索请求的时候，陷入局部最优，导致搜索返回的 K 个结果中属于请求的真实 K 近邻的比例很少。我们称之为由于部分节点被标记为删除导致的返回结果命中率低。

图 3-4展示了搜索过程遇上被删除节点，导致陷入局部最优而提前终止，最终使得返回结果命中率低的情况。在这里，被删除节点将原图拆分成了两个联通子图，极大降低了原图的联通性。

事实上，局部最优是不可避免的，这是因为 HNSW 算法构建的每一层图都只是近似 K 近邻图。但是部分节点被删除却加剧了这种趋势，由于被删除节点破坏了图的连通性，导致搜索过程中许多可走的路线被截断。此外，这种局部最优的情况在实践中是很难被发现的。这是因为一方面，由于节点删除是局部性的，很可能只有部分请求的结果才会受到被删除节点的影响。另一方面，由于对于实际的请求，我们很少会去主动评测每个请求的召回率，该过程只在专用数据集上才会发生。综合以上两点，由于部分节点被删除导致的部分请求命中

率低的问题是一种实践中很难被发现的问题。只有从源头上阻止该问题的发生，才能从根本上解决部分请求返回结果命中率的问题。

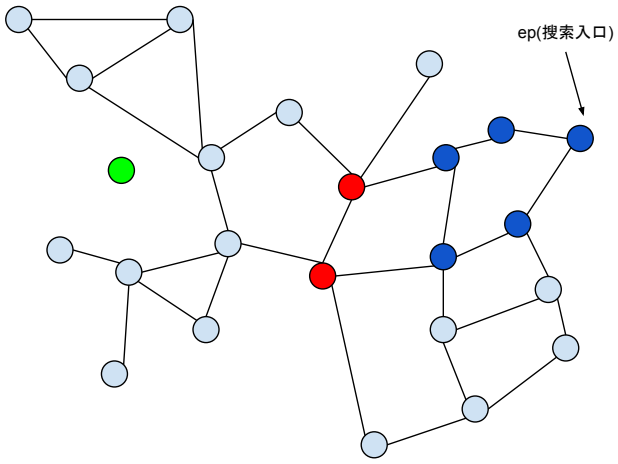


图 3-4: 搜索过程陷入局部最优，这里省略了边的方向。这里绿色节点是请求数据，红色节点是被删除节点，蓝色节点是搜索过程访问过的节点。可以看到，搜索过程在遇上红色节点之后停止了。

下一节我们将提出算法，正确处理 HNSW 构建的图中的节点删除问题。

3.2.3 节点删除算法

前面讨论了常用节点删除方式，即直接将节点标记为删除，以及该节点删除方式会导致的返回结果数量不足或者返回结果命中率低的问题。为了解决上述问题，本文提出了新的节点删除方法。

节点删除的的关键难点是每个节点只知道其出度边对应的邻居节点，而不知道其入度边对应的邻居节点。如果每个节点都保存其所有邻居节点的信息，会带来很大的存储和计算开销，是不可取的。为了获得有哪些节点拥有包含终点为被删除节点的信息，本文认为可以采用不需要额外存储开销的计算方式就可以获得尽可能多的这些节点的信息。

回忆节点的插入过程，会在每一层运行算法 3.1，获得一批节点作为新插入节点的邻居的候选者，然后从中选取部分节点作为新节点的邻居。同时，这些被选中的邻居也会将新节点作为其邻居的候选者。对于每个节点，由于其邻居数量是固定的，所以会先获得一批候选者，然后采用 SELECT-NEIGHBORS-HEURISTIC [13] 从候选者中选取部分节点作为邻居。通过这里我们发现，可以同样采用搜索算法 3.1，以某个将要被删除的节点作为搜索目标，在搜索过程中

记录有哪些节点包含目标节点作为邻居。同时，我们可以设置较大的 ef 以获得尽可能多的邻居信息。基于以上想法，我们设计了新的节点删除算法。由于我们新提出算法的核心思想是被删除节点与其他节点相互之间都需要删除相关信息，因此我们称新提出的节点删除算法为 HNSW Mutal-Remove，算法细节如算法 3.2所示。

算法 3.2 HNSW Mutual-Remove 的具体过程。

1: **function** HNSW-MUTUAL-REMOVE($d, d\_level, ep, ep\_level, ef$)

输入:

d 待删除的节点
d_level 待删除的节点的最大层次
ep 整个图结构搜索的入口节点
ep_level 入口节点的最大层次
ef 搜索过程中回溯队列的最大长度

输出:

删除节点 d 之后的图

2: **while** $ep\_level > d\_level$ **do**
3: $Ans \leftarrow$ SEARCH-LAYER($ep\_level, ep, 1, d$)
4: $ep \leftarrow$ nearest from Ans
5: $ep\_level \leftarrow ep\_level - 1$
6: **end while**
7: **while** $ep\_level > 0$ **do**
8: $Ans \leftarrow$ SEARCH-LAYER(ep_level, ep, ef, d)
9: **for** $x \in Ans$ **do**
10: **if** $d \in neighborhood(x)$ **then**
11: $neighborhood(x) \leftarrow neighborhood(x) - \{d\}$
12: **end if**
13: **end for**
14: $ep \leftarrow$ nearest from Ans
15: $ep\_level \leftarrow ep\_level - 1$
16: **end while**
17: $coordinate(x) \leftarrow$ inf

HNSW 节点删除算法 HNSW Mutual-Remove 的第 11 步会修改节点的邻居信息。这里并没有调用邻居选择算法 SELECT-NEIGHBORS-HEURISTIC。这是因为这里只涉及到节点的删除，并没有新增节点。此外，算法最后一步中将被删除节点的坐标设置为了无穷远处。这样设置有许多好处，下面进行介绍。

有利于判断一个节点是否已经删除。许多时候我们会需要检查某个节点是否被删除了，直接将被删除节点的坐标设置为无穷远是一种非常方便实用的方法，因为这种方法除了方便查询某个节点是否被删除，并不需要额外的存储开销。

有利于图的动态更新。我们知道，HNSW 算法在构图的时候，每个节点的最大邻居数量是固定的。通常情况下，最底层的节点的最大邻居数量为 $2*m$，而其他层的最大邻居数量为 m，这里的 m 是 HNSW 的重要构建参数之一。前面提到，为了维持固定数量的邻居，会采取邻居选择算法 SELECT-NEIGHBORS-HEURISTIC 从一批候选者中选取部分节点作为邻居。具体的，有两个过程会进行邻居的选取。

第一个是，当新节点被插入的时候，会从算法 3.1返回的 $efConstruction$ 个候选者中选取部分节点作为邻居。

第二个是，当某个节点被当作新插入节点的邻居，而该节点的邻居数量已经达到其最大邻居数量的时候，会将该节点的旧邻居以及新插入节点作为候选者，采用邻居选择算法选择一批节点作为该节点的新邻居。算法 SELECT-NEIGHBORS-HEURISTIC 的核心思想是，邻居的选取不能仅仅看距离的远近，还要考虑节点的方向性。比如，对于某个节点 q 的某个邻居候选者 c，只有当节点 c 相对于目前已经选中的节点是离 q 最近的节点，才会选中节点 c 作为 q 的邻居。

可以看到，该过程会使得位于无穷远处的节点无法被选中作为邻居。这样，随着新的节点的插入，那些“残留”的没有被完全删除的节点会随着现有节点的动态更新而被移出邻居列表。给图带来了自愈能力。

被删除节点在搜索过程中会被忽略。由于被删除节点的坐标位于无穷远，这离我们搜索距离目标最近的点的原则是背道而驰的。因此，在搜索过程中这些位于无穷远处的节点就不会被引入结果队列，从而影响最终的搜索结果。这也是符合实际需求的，这是因为对于被删除节点的预期就是它不会对搜索结果产生任何影响，它们应当是透明的。

HNSW 节点删除算法 HNSW Mutual-Remove 的复杂度分析：算法的核心步骤是的将被删除节点作为目标节点进行搜索。其他过程，如邻居选择，将节点坐标置为无穷大等均可以认为只消耗常数时间。而这个搜索过程的复杂度处于 $O(log(N))$ 这个量级。当然，$efSearch$ 参数设置地越大，加在该复杂度上的常数系数也越大。这样节点删除过程拥有和搜索过程相同的复杂度。这个复杂度在实践中是一个非常能够接受的速度，因为实践中搜索过程是十分快的。此外，节点的删除并没有引入额外的存储开销。

3.2.4 HNSW 节点更新算法

这里讨论的节点更新指的是更新节点的坐标，这些坐标也就是高维向量往往是某种特征向量，许多应用场景都需要对这些特征向量进行更新。我们知道，直接更新节点的坐标会破坏正常的图结构，但是基于 HNSW 节点删除算法 HNSW Mutual-Remove，可以轻松的实现节点的更新。具体的，节点的更新可以分为两步，第一步将待更新节点删除，第二步将节点作为新节点插入图中。由于这里核心过程都是基于 HNSW 节点删除算法，具体过程这里不再赘述。

3.3 实验与分析

本节将进行一系列实验，分别是对 HNSW 节点删除算法 HNSW Mutual-Remove 的实验、HNSW 与线性扫描的对比实验以及 HNSW 对聚类算法 k-means 的加速实验。实验中包含 HNSW 对 k-means 的加速实验是因为许多近似最近邻搜索算法如 IVF-HNSW 都需要用到 k-means 算法，而对 k-means 进行加速是一种实用的工程技巧，因此这里也会对其进行介绍。

HNSW 算法的主要参数前面已经介绍过，分别是控制构建索引质量的参数 $efConstruction$、控制搜索结果质量的参数 $efSearch$ 以及控制 HNSW 构建的图中每个节点最大邻居数量的参数 m。在每个实验中，我们都会给出这些参数的取值。

3.3.1 对节点删除算法 HNSW Mutual-remove 的实验

为了验证本文提出的 HNSW 节点删除算法 HNSW Mutal-Remove 的效果，本节设计了在不同规模的随机数据集上的实验。HNSW 原有的节点删除算法带来的问题主要体现在大量删除节点之后会存在搜索返回结果不足的问题，在实践中我们发现这种情况虽然发生的很少，但是一旦出现都会带来很严重的问题。为了验证提出的 HNSW Mutual-Remove 算法是否能够有效解决这个问题，我们通过在构建好的索引中大量删除节点，然后用大量请求数据来观察出现返回结果数量不足 K 的情况。

我们采用了三组不同规模的随机数据，在这三组数据的实验上，我们将请

求数据的数量设置为 10，000，返回结果数量 K 设置为 10，HNSW 的搜索参数 $efSearch$ 设置为 20。此外，由于数据规模都比较小，HNSW 的构建参数 m 也设置的比较小。这里为了保证能够观测到返回结果数量不足 K 的问题，我们将搜索数据的数量设置的比较大，保证了 HNSW 构建的图上每个区域的节点都能被访问到。在实现上，这里的实验都是基于 hnswlib 完成的，代码运行平台为 Apple M1。

表 3-1: 大量删除节点之后搜索返回结果数量的情况

数据数量与删除比例	数据维度与数据类型	HNSW 构建参数	是否使用 HNSW Mutual-Remove	返回结果不足 K 的次数
100,000 0.7	128 4 字节浮点数	$efConstruction = 200$ $m = 8$	否	44
			是	**0**
100,000 0.7	128 4 字节浮点数	$efConstruction = 200$ $m = 12$	否	0
			是	0
500,000 0.8	64 4 字节浮点数	$efConstruction = 200$ $m = 8$	否	53
			是	**0**
500,000 0.8	64 4 字节浮点数	$efConstruction = 200$ $m = 12$	否	12
			是	**0**
1,000,000 0.8	32 4 字节浮点数	$efConstruction = 200$ $m = 8$	否	61
			是	**0**
1,000,000 0.8	32 4 字节浮点数	$efConstruction = 200$ $m = 12$	否	13
			是	**0**

实验中其他参数设定以及实验结果如表格 3-1所示。从表格中的实验结果可以得到以下结论。

首先，我们观察到在三组不同规模的实验中，采用 HNSW Mutual-Remove 均能防止出现返回结果不足 K 的问题，即使被删除的数据的数量占比非常大。而采用默认节点删除方法（将节点标记删除）都或多或少产生了返回结果不足 K 的问题。这验证了算法 HNSW Mutaul-Remove 能有效避免出现返回结果数量不足 K 的问题。

此外，在每组对比实验中，我们分别将 HNSW 的构建参数 m 设为 8 和 12。我们发现，增加参数 m 能有效抑制默认节点删除算法导致返回结果不足的问题，这是因为返回结果数量不足问题产生于算法 3.1在搜索过程中陷入局部最优。增加 m 的大小增加了每个节点的出度和入度，这样即使部分边被标记为不可用，仍然能够有可用的边。但是由于增加参数 m 会增加构建的索引的大小，

而 HNSW Mutual-Remove 并不需要额外的存储开销，这也进一步凸显了 HNSW Mutual-Remove 的优势。

3.3.2 线性扫描与 HNSW 的对比实验

表格 3-2列出了 HNSW 算法和线性扫描的复杂度对比。由于 HNSW 算法在高维数据上的确切复杂度并没有经过严格的数学推导和实验验证[13]，所以上述表格中关于 HNSW 的复杂度只是一个非常粗略的结果。此外 HNSW 算法的各个过程的实际运行时间还与具体参数息息相关，也就是 HNSW 算法复杂度的常数系数有时候会很大，对整个算法的性能影响也会很大。

本文首先设计实验来探讨 HNSW 算法和线性扫描在不同规模以及不同维度的数据上的性能差异。由于线性扫描的召回率总是 1，但是 HNSW 几乎很难接近 1，在实验设定上我们将比较 HNSW 在固定召回率上的表现与线性扫描进行比较。比如固定召回率为 0.99，比较 HNSW 算法和线性扫描的搜索效率。通常情况下，达到 0.9 甚至 0.8 以上的召回率就能够满足许多应用的需求。

表 3-2: 线性扫描和 HNSW 复杂度对比

	预处理复杂度	新数据插入复杂度	搜索复杂度
Linear Scan	O(0)	O(0)	O(n)
HNSW	O(nlog(n))	O(log(n))	O(log(n))

为了充分对比 HNSW 算法和线性扫描算法，本文从多个角度设计了对比实验。首先是比较在固定召回率的情况下，HNSW 算法和线性扫描算法的召回时间和数据量之间的关系。由于 HNSW 算法在不同召回数量 K 的情况下下平均召回时间差异较大，本文采用在不同的固定召回数量 K 的条件下对比两个算法。本文接着在人造数据集上对比不同维度上两个算法的性能差异。最后本文接着将 HNSW 算法与基于 GPU 进行加速的线性扫描算法进行比较，这是由于 GPU 能够对大量数据的并行运算进行加速，同时线性扫描算法本身适用于采用 GPU 进行并行加速。

在不同规模的数据上的实验。该实验在近似最近邻搜索常用数据集 ANN_SIFT1M 以及 ANN_GIST1M [21] 上展开，前者是 128 维的 SIFT[40] 特征，后者是 960 维的 GIST[41] 特征。我们将 HNSW 算法的召回率设置为 0.95，选取 0.95 是因为这是一

表 3-3: HNSW 与线性扫描对比实验参数设定

数据集	m	$efConstruction$	请求数据数量	CPU 数量	GPU 数量
ANN_SIFT1M	8	50	64	8	0
ANN_GIST1M	48	460	64	8	0
RAND_DIM	48	200	100	1	0
RAND_NUM	48	200	128/8192	16	5

个高召回率场景的常用选择。即在 HNSW 在达到 0.95 的召回率下将其与线性扫描进行对比。这里实验中 HNSW 算法和线性扫描的实现采用了 faiss 库[①]的实现。实验平台选用的 CPU 型号为 Intel(R) Xeon(R) Gold 6248，最高频率为 2.50GHz。所有实验都开启了 OpenMP 加速，且 OpenMP 可使用的最大核数设置为 8。

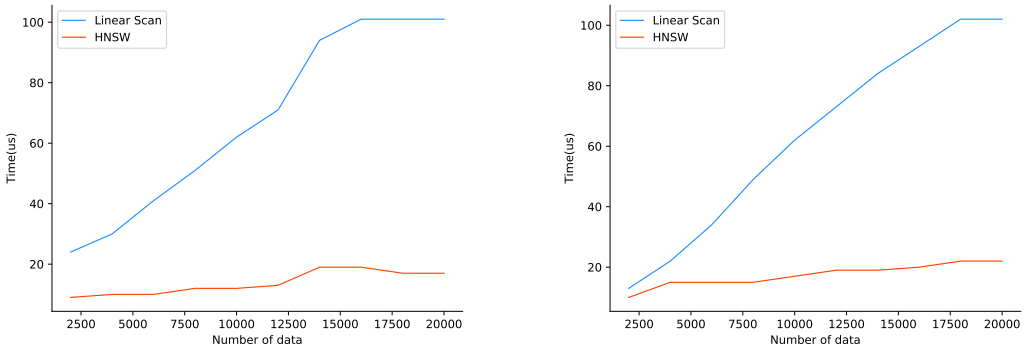
我们在每个数据集上分别进行了四组实验，对应 K 的取值分别为 1，10，50，100。HNSW 在数据集 ANN_SIFT1M 以及 ANN_GIST1M 上的构建参数如表格 3-3所示。对于 HNSW 的搜索参数 $efSearch$，我们的设置方式为从 K 开始递增 $efSearch$，直到 HNSW 的召回率达到 0.95。

在 ANN_SIFT1M 上的实验结果如图 3-5所示，ANN_GIST1M 上的实验结果如图 3-6所示。这些实验显示，无论是在较低维度还是在更高维数据上，HNSW 相对于线性扫描都具有决定性的优势。在这两个实验中，HNSW 在数据数量很少的时候，从 2000 这个数量级的数据开始，HNSW 就开始远远甩开线性扫描。此外我们观察到到更大的召回数量 K 能让线性扫描更“容易”接近 HNSW。这是由于更大的召回数量 K 会让 HNSW 进行更多的距离计算，访问更多数据。极端情况下，当 K 取值为整个数据集的数据数量的时候，HNSW 算法就退化为线性扫描了。

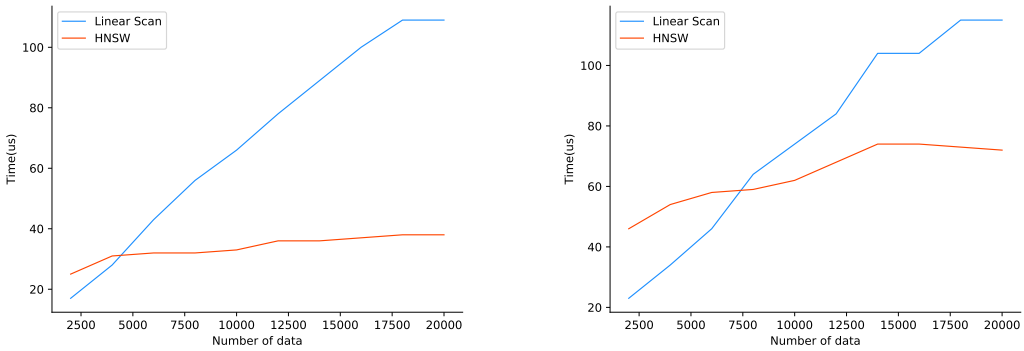
下面进一步探索数据维度对搜索效率的影响。

在不同维度数据上的实验。维度灾难问题催生了近似最近邻搜索。对于线性扫描来说，搜索时间/计算量是同数据维度正相关的。计算复杂度为 O(dn)。对于 HNSW，其在高维数据上的搜索复杂度并没有经过很好的实验验证，这是因为验证实验需要极大的计算资源[13]。同时，高维数据对 HNSW 的构图过程会产生什么影响也未可知。但是总的来说，更高的维度让 HNSW 构图变得更加困难。这里的困难是指，是否更高维度的数据使得图的搜索过程变得更加难以收敛到请求数据附近，从而增加计算量，甚至召回率的提升也会变得更加困难。但是从

[①]https://github.com/facebookresearch/faiss/



(a) $K = 1$ (b) $K = 10$



(c) $K = 50$ (d) $K = 100$

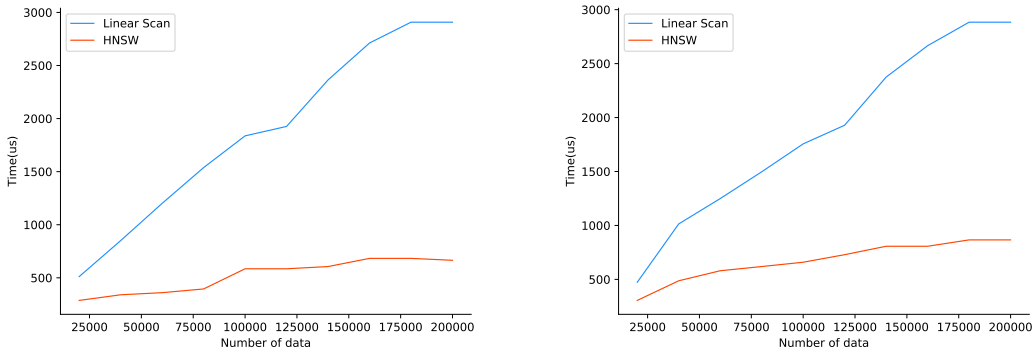
图 3-5: ANN_SIFT 数据集上，给定召回数量 K，线性扫描和 HNSW 算法的平均召回时间与数据集规模之间的关系。这里曲线位置越低证明搜索速度更快，对应算法也就更好。

另一个角度来看，高维数据让每一次距离计算都变得非常耗费计算资源，HNSW 能大幅减少节点之间的距离计算，在高维数据上 HNSW 的优势会更加明显。

在不同数量的数据集上的实验结果显示，HNSW 在数千数据量的时候就已经超越线性扫描，后续更是大幅领先。因此这里的实验选取数千至数万的数据进一步探索不同维度下 HNSW 算法与线性扫描的性能表现。

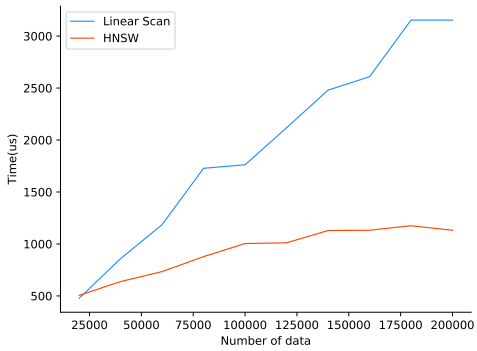
我们将 HNSW 的召回率设定为 0.95，并让 $efSearch$ 自 K 开始依次递增，直到达到给定召回率。在实验中，我们让召回数量 K 大小分别固定为 10 和 40，然后让数据维度从 2 依次递增至 100，进而对比 HNSW 和线性扫描的平均召回时间。由于本实验需要递增数据维度，所以采用了随机数据，我们记为 RAND_DIM。HNSW 在这一系列数据集上的构建参数如表格 3-3所示。

本实验使用的平台 CPU 型号为 Intel i5-7500，CPU 最高频率为 3.40GHz, 在这一系列实验中我们只使用了单个核，这是因为该实验平台总 CPU 数量很少，只有四个核，使用多个核会发生频繁的 CPU 切换，使得实验结果波动很大。所

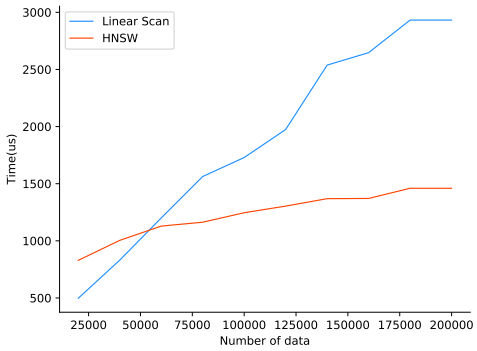


(a) $K = 1$

(b) $K = 10$



(c) $K = 50$

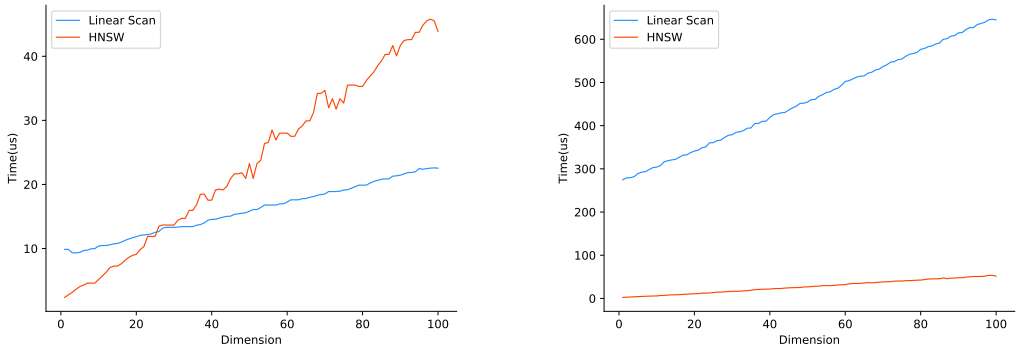


(d) $K = 100$

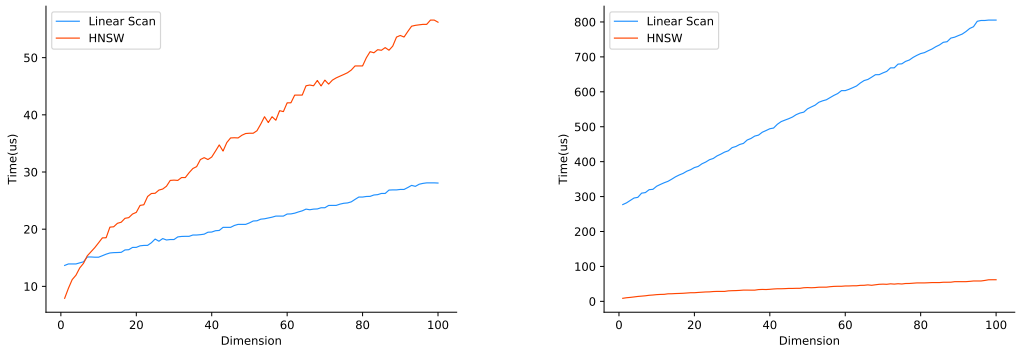
图 3-6: ANN_GIST 数据集上，给定召回数量 K，线性扫描和 HNSW 算法的平均召回时间与数据集规模之间的关系。曲线位置越低效果越好。

以为了实验结果更加美观，这里只使用了单个 CPU 核。

我们分别在数据数量为 5,000 以及 200,000 的随机数据上进行了实验。实验结果如图 3-7所示。从图中可以得出两个结论。第一，无论是线性扫描还是 HNSW 算法，它们的搜索时间都随着搜索搜索维度线性增长，这一实验结果说明 HNSW 算法和线性扫描一样，对于同等规模的数据，搜索复杂度会随着数据维度线性增长。第二，不同规模的数据下，HNSW 算法和线性扫描算法搜索时间随着维度增长的快慢不同，也就是斜率不同。我们看到，当数据规模比较小的时候，HNSW 算法增长速度更快，而当数据规模更大的时候，线性扫描的增长速度更快。我们猜测这是由于 HNSW 算法的额外开销带来的，虽然 HNSW 算法能够极大减少节点之间的距离计算次数，但是当数据规模比较小的时候，这一点并不不能显示出很大的优势。反而，HNSW 算法的额外开销使其在小规模数据上并没有明显优势。这里 HNSW 算法的额外开销包括，搜索过程中维护一个大小为 $efSearch$ 的优先级队列，随机的内存访问开销（线性扫描是连续的内



(a) $K = 10$，数据量 5,000　　(b) $K = 10$，数据量 200,000



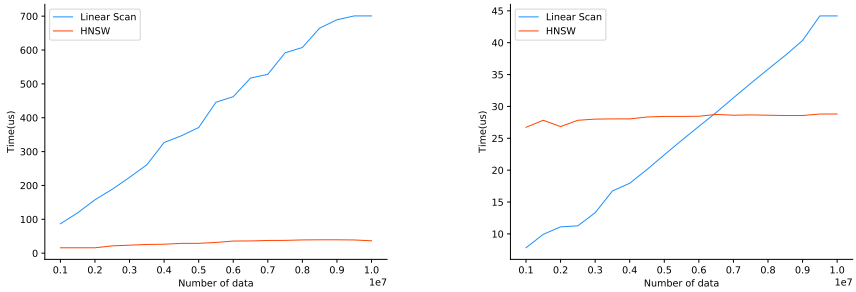
(c) $K = 40$，数据量 5,000　　(d) $K = 40$，数据量 200,000

图 3-7: 不同维度下 HNSW 与线性扫描的对比。这里每条曲线是平均召回时间与数据维度的关系，曲线位置越低效果越好。

存访问，开销更低）等等。由于我们采用让 $efSearch$ 从 K 开始递增的方式，我们决定采用不同的 K，来观察优先级队列大小对上述的斜率的影响。对比 3-7a 和 3-7c 我们可以发现，由于优先级队列更大了，HNSW 算法的搜索耗时更快的超越了线性扫描。这也证实了我们的分析，即维护搜索时的优先级队列是一项重要的额外开销。

通过对比不同规模、不同数据维度、以及不同搜索数量 K 下线性扫描与 HNSW 算法的性能表现。我们发现，除非数据规模特别小，比如数据只有几千条，HNSW 算法的效果都要远远好于线性扫描。只有在数据规模很小的时候，线性扫描才会展现出优势。总的来说，在需要近似最近邻搜索的场景下，HNSW 算法是很好的选择。对于 HNSW 算法，其使用简单，只需要设置好控制节点之间连边数量的参数 m 即可。通过一系列实验，结合 HNSW 算法官方给出的建议[13]，m 一般在 8 至 48 之间取值即可，数据规模越大，其取值也要相应增大。

在 GPU 上的实验。下面进行基于 GPU 进行加速的线性扫描与 HNSW 的对



(a) 请求数据数量为 128 (b) 请求数据量为 8192

图 3-8: 不同数据规模下基于 GPU 加速的线性扫描和 HNSW 算法搜索效率对比。这里的曲线是平均搜索时间与数据数量的关系，曲线位置越低效果越好。

比实验。实验开始前首先介绍并行计算技术。包括单指令多数据技术，基于图形处理单元进行并行化计算等。单指令多数据技术（SIMD）指的是用单个指令操作多个数据完成运算。比如，在近似最近邻搜索问题中，最大的计算量花费在了点之间的距离计算上。每一个点都是空间中的一个向量，假设这些点位于 d 维空间，如果需要计算两点之间的欧式距离，则需要对每个维度重复相同的操作，即不断的做减法与乘法，最后将各个维度差的平方进行求和。SIMD 技术能够让我们单个加法能够操作多个数据，比如执行一次减法指令就能够将多个维度的数据之差计算出来，我们可以理解维 SIMD 技术是一种将执行向量间运算的技术。虽然 SIMD 技术使得单个指令能够操作多个数据，从而达到加速的目的，但是单个指令能够操作的数据是有限的，比如 Intel 公司的 SIMD 运算指令集 AVX2 最高只支持 512 个比特的数据[42]。各大处理器厂商对 SIMD 技术的支持都是通过扩展指令集完成的，即将将各种 SIMD 处理器和 CPU 整合来扩大 CPU 的并行计算能力。前面所有实验都是在 CPU 上完成的，当然也都利用到了各个 CPU 平台上的 SIMD 技术。

通过前面不同规模以及不同维度下线性扫描和 HNSW 算法的对比实验，我们发现基于 CPU 的 SIMD 并行技术并不能弥补两个算法搜索速度之间的鸿沟。所以我们想采用其他并行化技术来对比线性扫描和 HNSW 算法。而得益于计算机硬件的快速发展，GPU 进行并行运算已经成为并行计算的新趋势。目前 GPU 运算已经广泛应用于各种深度学习任务中，在各种实际应用如视频游戏也应用广泛。一般来说，GPU 适合于满足如下条件的应用[7]:

计算量非常大。比如大型视频游戏每秒需要完成数百数百帧画面的渲染，近似最近邻搜索中线性扫描每次都需要计算每个请求数据和所有数据的距离。

需要大量的并行化。换句话说，就是应用中很大一部分都是可并行化的。比如视频游戏中一帧画面的不同部分可以独立并行进行渲染。近似最近邻搜索中，线性扫描中可以同时计算请求和所有数据的距离，但是 HNSW 算法中就无法达到这么大的并行度，这是因为 HNSW 搜索过程中只会访问一部分节点，而且后续所能访问的节点还依赖于前面访问过的节点。所以，HNSW 算法的搜索过程的并行度并不高。

吞吐量比响应速度更加重要。GPU 的设计原则之一就是强调系统的吞吐量，它的单次运算需要用时成百上千个时钟周期。而且 GPU 运算的流水线是没有分支预测的负担的，进一步提高整体的吞吐量。

从上面可以看出，GPU 适合用于需要进行大量并行计算的任务，同时任务的吞吐量要求比延时要求要高。由此我们得出在近似最近邻搜索问题中，线性扫描十分适合于采用 GPU 进行加速，而 HNSW 算法搜索过程中，只会访问一部分节点，而只有在确定了前面已经访问过的节点之后才能确定后面需要访问的节点，这样使得 HNSW 算法不适合于利用 GPU 进行加速。另一方面，HNSW 算法需要频繁地随机访问内存，而让 CPU 把数据拷贝到 GPU 是个很慢的过程。所以，HNSW 算法不适合于用 GPU 进行加速。最后是响应速度的吞吐量的问题，虽然相对于 CPU 来说，GPU 的响应速度比较慢，但是经过实验测算，对于数百万量级的高维数据，每次搜索只包含一个请求的条件下，GPU 也能在数毫秒时间内完成搜索，这对于绝大多数实际应用来说是足够的。此外，绝大多数应用中，可以将请求“打包”起来一起搜索，借助于 GPU 进行运算，可以获得很大的吞吐量。

为了更加科学直观的了解 GPU 的加速效果。我设计了两组实验来对比基于 GPU 进行加速的线性扫描和利用 CPU 进行运算的 HNSW 算法。这两组实验分别设定为拥有低吞吐量（单次请求数据较少）和高吞吐量（请求数据较多）。每组实验分别测定两个算法在不同规模（从一百万到一千万）数据集上，线性扫描和 HNSW 算法的平均搜索耗时。我们采用了随机数据集，数据维度为 128，数据集名字记为 RAND_NUM，实验中相关参数设定见表格 3-3。本实验的 CPU 型号为 Intel(R) Xeon(R) Gold 6248，最高频率为 2.50GHz，GPU 型号为 Tesla V100 SXM2 32GB。

实验结果如图 3-8所示。从图中可以得出两个结论。第一个是 GPU 的强大

计算能力弥补了 HNSW 算法和 HNSW 算法之间的巨大差距。前面的实验显示，线性扫描只有在非常小的规模上上相对于 HNSW 算法才有几乎可以忽略的优势，即 HNSW 算法几乎总是优于线性扫描。但是，基于 GPU 加速的线性扫描已经拥有了可以与 HNSW 匹敌的搜索速度。从数百万到上千万的数量级的高维数据上，线性扫描的平均耗时还不到 1 毫秒，这样的吞吐量证明基于 GPU 加速的近似线性扫描完全适合于大规模数据的精确最近邻搜索。如果有足够的 GPU 资源，或者对准确率要求极高，应当采用基于 GPU 加速的线性扫描。第二个是请求数据越多，线性扫描的吞吐量越大。我们看到，在请求量很大的情况下，线性扫描的平均搜索延时已经处于和 HNSW 一样的水平，如图 3-8b所示。由于请求越多，越能利用 GPU 的高吞吐量能力，实际应用中，我们应当尽可能将请求“打包”，批量放入 GPU 进行搜索。

通过前面一系列分析与实验，我们验证了 HNSW 算法作为目前综合表现最好的近似最近邻搜索算法的优异性能。并得出结论，在任何时候，应当优先考虑 HNSW 算法。此外，我们也证实了 GPU 对线性扫描拥有巨大的加速能力，在 GPU 计算资源丰富或者对召回率要求百分之百的场景，可以考虑基于 GPU 加速的线性扫描。

3.3.3 HNSW 加速 k-means 训练速度的实验

本节简要介绍 HNSW 算法对 k-means 算法的加速效果。我们知道 k-means 算法运行过程中需要迭代许多轮次，在每个迭代轮次中，每个数据都需要搜索距离其最近的聚类中心。对于 n 个 d 维数据的，聚类个数为 k 的聚类过程，每次迭代的时间复杂度为 $O(ndk)$。对于一些 k 取值比较大的大规模数据聚类过程，k-means 是非常耗时的。这个时候可以采用 HNSW 来索引 k 个聚类中心，加快每个数据搜索距离其最近的聚类中心的过程。

我们采用随机数据的实验来验证这一效果，实验采用的硬件平台包含四个型号 Intel i5-7500，最高频率为 3.40GHz 的 CPU。在实验中，每次迭代结束之后，对于新计算得到的 k 个聚类中心，会重新使用 HNSW 构建索引，供下一轮迭代使用。表格 3-4展示了在随机数据集上的结果。结果显示当训练轮次达到 10 轮的时候，两个算法的损失函数值都是接近的，但是采用 HNSW 优化后的 k-means 算法的运行速度大幅优于未优化的 k-means 算法。

表 3-4: 采用 HNSW 优化后的 k-means 算法与原始算法的对比结果

是否采用 HNSW 进行优化	数据数量	数据维度与数据类型	k 的大小	迭代轮次	训练时间（s）	最终误差（10^6）
否	1,000,000	64 维，4 字节浮点数	10,000	10	735	4.91
是	1,000,000	64 维，4 字节浮点数	10,000	10	**63**	4.95

这里之所以介绍 HNSW 算法对聚类算法的优化措施，是因为近似最近邻搜索常用的算法如倒排索引都需要预先对数据进行聚类，一般对于大规模的数据，聚类的参数 k 的取值也会变得很大。这个时候可以采用 HNSW 来优化 k-means 的训练速度。由于该方法更加接近于是一种工程技巧，而且比较简单，这里就不再深入进行介绍。但是我在实践中发现，该技巧不失为一种简单而又有效的方法。

3.4 本章小结

本章给 HNSW 提出了新的节点删除算法 HNSW Mutual-Remove 算法，解决了 HNSW 大量删除节点导致部分搜索请求返回结果数量不足 K 的问题。我们的实验也进一步验证了 HNSW Mutual-Remove 算法的有效性。此外，通过 HNSW 与线性扫描算法的对比实验，我们发现基于 GPU 进行加速的线性扫描算法拥有接近 HNSW 的性能。在对召回率要求极高的场景应当优先选用基于 GPU 加速的线性扫描算法。最后，本文通过实验验证了可以将 HNSW 应用于 k-means 聚类方法，加快大数据下聚类的速度。

第四章　近似最近邻算法的索引压缩

虽然 HNSW 算法性能优越，但是当数据规模变大，以 HNSW 为代表的 ANNS 算法存在索引模型内存占用大的问题。本章以 HNSW 为例分析了 ANNS 算法内存占用大的原因，并提出可以分别从压缩原始高维向量以及采用更加轻量级的架构来组织数据这两个方向减少索引模型的大小。前者一般采用标量量化与矢量量化对高维向量进行压缩，后者可以使用倒排索引等结构来组织数据。IVF-HNSW 虽然结合了倒排索引与量化方法，能够对索引大小进行大幅度的压缩，但是它在大规模数据上存在索引构建速度慢的问题，无法满足实际应用需求。本文针对这个问题提出了 Balanced IVF-HNSW，它成功解决了 IVF-HNSW 索引构建速度慢的问题，同时能够保持优秀的召回效果。

4.1　近似最近邻搜索算法在内存占用上存在的问题

近似最近邻算法通过某种数据结构将需要检索的高维向量组织在一起，最终得到的数据结构就是索引。算法运行过程中索引被放在内存中。而由于，在大规模数据上构建的索引都比较大，因此最终的内存资源消耗也很大。下面以目前主流方法 HNSW 为例，对索引的内存占用问题进行深入剖析。

4.1.1　索引为何需要占用大量内存

首先，近似最近邻搜索算法都是通过牺牲召回率/准确率来换取时间。对于每一个请求，算法的搜索过程都只会访问 n 个数据中的部分数据。由于事先不知道每次请求需要访问哪些数据，所以需要将所有数据都存下来，以方便计算请求数据与任意一个被索引的数据之间的距离。这里就牵涉到一个问题，这些被索引的数据存储到哪儿。一般来说有两个选择，存储到廉价的硬盘上或者存储在内存中。我们知道，硬盘与内存的一个重要区别是访问速度，内存的读写速度大幅优于磁盘，内存的访问速度可以达到 10GiB/s，而截止 2017 年磁盘的最高访问速度约为 2000MB/s[43]。到这里我们知道，如果数据量越多，占用的存储

开销也就越高，比如，10 亿条 128 维的数据，假定每个维度的数据都用四个字节的浮点数表示，那么需要的存储空间约为 500GiB。一般来说，单个机器很少拥有这么大的内存。

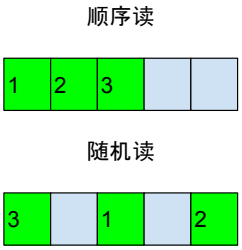


图 4-1: 顺序读与随机读。这里编号 1，2，3 表示按照时间顺序需要读的数据。

另外一个问题就是，无论数据是存储在内存还是磁盘上，在搜索的时候读这些数据的方式都是随机读。这是因为每次搜索的时候，都只会访问一部分数据，这种情况下，无论以什么方式组织所有数据，都无法让每次请求对数据的访问变成顺序读。这里的随机读与顺序读的区别如图 4-1所示。顺序读场景中，先后要读的数据 1、2、3 正好是连续存放着的，随机读场景中，先后要读的数据，1、2、3 并不是连续存放在一起的。如果数据是放在内存中，由于局部性原理，随机取内存会导致频繁的 cache miss [44]，进而降低程序运行效率。如果数据是放在磁盘上，那么随机读每次都有磁盘寻道以及读数据的开销，与此同时内存也会有频繁的发生 cache miss。这里对磁盘的寻道会花费大量的时间。而如果是顺序读，内存发生 cache miss 的概率会变低，磁盘也只需要进行一次寻道和一次读取数据。总的来说，在搜索过程中，对索引中数据的读取方式是随机读，这种读取数据的方式具有很大的时间开销。

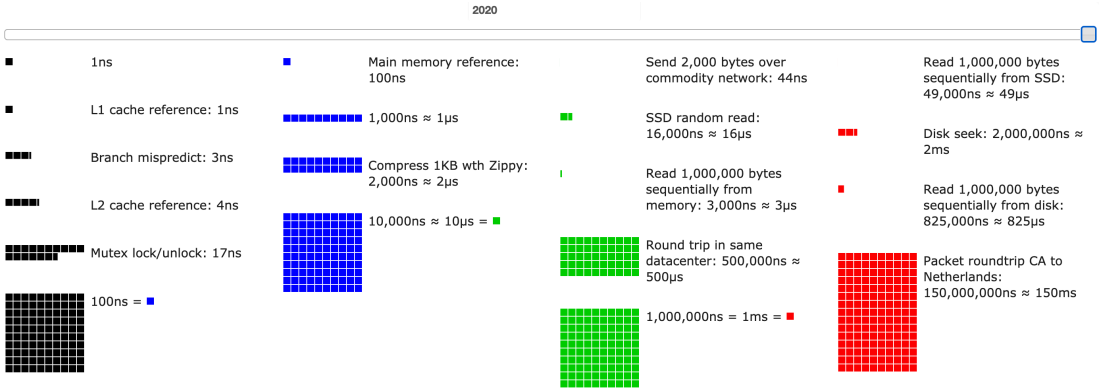


图 4-2: 最新（截至 2020 年）各种存储设备的访问延时[②]

[②]https://colin-scott.github.io/personal_website/research/interactive_latency.html

通过以上分析我们知道了两点，第一是近似最近邻搜索必须存储所有的数据，以计算请求数据与每个数据之间的距离；第二是，搜索的时候对数据的访问是随机读，磁盘的随机读相对内存来说开销更大。下面结合具体数据分析二者的差距有多大。如图 4-2 所示为截至 2020 年最新的 CPU 访问内存和磁盘需要花费时间的具体数据，我们可以看到单次访问内存的时间消耗是 0.1us，而每次对磁盘的随机读操作都需要进行寻道，而这种寻道每次都需要花费 2ms 的时间，此外从磁盘读取一次数据也要花费数百 us 的时间，因此相较于访问内存，访问磁盘慢了一个数量级。虽然访问硬盘相对于访问内存来说会显得很慢，但是如果它能够满足用户需求，那么也是可以接受的，因为我们的算法最终是要服务于应用的，脱离了实际应用的算法是没有价值的。

根据本文章作者的经验，实际场景中，对于每个请求，搜索需要的时间不超过 10ms 是比较满足应用需求的。对于一个百万的数据集，我们假设采用近似最近邻算法对数据进行索引之后，每次请求最多需要访问 1%（后面我们对 HNSW 算法的实验表明这个比例是一个比较合理的上限）的数据，即 1 万条左右的数据。那么就就需要 1 万次对内存或者磁盘的随机访问。对于内存来说，1 万次对内存的访问加起来需要大约 1ms 的时间，加上传送数据以及计算所需要的时间，HNSW 是能够在 10ms 完成每次请求的。而这也与实践中使用 HNSW 的结果是符合的，在实践中，HNSW 算法构建的索引几乎总能够在数毫秒内完成搜索。而对于磁盘来说，1 万次随机的内存访问就需要 1 万次寻道，仅仅花费在寻道上的时间就需要 20s 的时间，显然这个结果是无法满足实际应用需求的。

以上通过分析说明了为什么对于大规模索引需要占用大量内存：我们需要存储全部数据且只有将数据放在内存才能满足实际应用对搜索速度的要求。此外，索引本身还有独立于数据的附属数据结构，也会带来内存的开销。下面通过对 HNSW 算法的具体分析来说明这一问题。

4.1.2 对 HNSW 算法的内存占用分析

记 HNSW 构建的索引大小为 S_{index} 字节，其中数据部分以及附属数据结构占用存储空间分别为 S_{data} 字节和 $S_{structure}$ 字节，则有

$$S_{index} = S_{data} + S_{structure}, \tag{4-1}$$

其中，

$$S_{data} = n * d * d_{size}, \tag{4-2}$$

这里，n，d，d_{size} 分别表示被索引数据的数量、维度以及每个维度占用的字节数。HNSW 算法的附属数据结构为一个有向图。每个节点都需要保存它的邻居节点的信息。由于每个节点的最大层数都是随机的，我们需要使用其平均高度来分析每个节点占用的存储空间。根据 HNSW 算法的默认参数设定[13]，节点的平均层数，即节点层数的期望为

$$L_{average} = \frac{1}{\log m} + 1 \tag{4-3}$$

这里的 1 表示无论如何每个节点至少存在于最底层的图中。章节 3提到，参数 m 为第一层以上每层图结构中每个节点的最大邻居数量，而最底层图结构中每个节点的最大邻居数量为 $2 * m$。那么，每个节点为了存储其邻居信息需要的字节数为

$$S_{node} = id_{size} * (2 + \frac{1}{\log m}) * m \tag{4-4}$$

这里，id_{size} 表示用来存储每个节点的 ID 需要占用的字节数。由于需要用 ID 区分不同的节点，常用做法是按照节点添加的顺序给每个节点赋予一个从零开始递增的序号作为 ID。通常情况下，只需要 4 个或者 8 个字节就能够满足需求。这样，根据式 4-2以及式 4-4就可以得到索引的总大小（单位为字节数）：

$$\begin{aligned} S_{index} &= S_{data} + S_{structure} \\ &= S_{data} + n * S_{node} \\ &= n * d * d_{size} + n * id_{size} * (2 + \frac{1}{\log m}) * m \\ &= n * \left(d * d_{size} + id_{size} * (2 + \frac{1}{\log m}) * m \right) \end{aligned} \tag{4-5}$$

根据以上推理，数据维度 d 和 HNSW 构建参数 m 是决定索引大小的关键因素。一般来说，d 的取值因数据而异，变化范围从数十到数千。而参数 m，前面提到它的取值在 5 到 48 是一个比较合理的范围。因此，对于较高维度的数据，数据维度对索引大小的影响更大。而当维度并不是很大的时候，参数 m 也会对

最终索引大小产生重要影响。对于 HNSW 来说，m 决定了图结构的存储开销，数据维度决定了数据的存储开销。

表 4-1: HNSW 在不同数据集上构建的索引大小

	数据条数	数据维度 d	参数 m	索引大小（MB）	数据占索引大小比例（%）
ANN_SIFT1M	1,000,000	128	16	625.6	78.0
			32	756.8	64.5
ANN_GIST1M	1,000,000	960	16	3,799.4	96.4
			32	3,930.7	93.2
ANN_SIFT1B	1,000,000,000	128	16	259,399.4	47.1
			32	390,625.0	31.2
RAND1B	1,000,000,000	1280	16	5,020,141.6	97.3
			32	5,281,808.9	92.4

下面通过几个具体数据集构建完成的索引大小来直观展示数据量，数据维度，参数 m 等因素对索引大小的影响。表格 4-1展示了在 ANN_SIFT1M，ANN_GIST1M 以及 ANN_SIFT1B[45] 数据集上所构建的索引大小，此外我们添加了一个人造数据集 RAND1B。所有数据的 ID 均使用四个字节表示，除了 ANN_SIFT1B 数据集中每个维度的数据用一个字节表示，其他的数据集每个维度的数据均为四字节的浮点数。从表格中的结果我们发现对于较低维度（一百维左右）的数据，HNSW 算法特有的图数据结构的存储开销是不可忽视的，比如在 ANN_SIFT1B 的实验结果中，图结构的存储开销超过了数据本身的开销。但是对于更高维度的数据，图结构本身的存储开销就显得有点微不足道了，而且随着数据量的增加，高维数据构建的模型会变得非常大，单个机器难以承载这么多的数据。比如 RAND1B 数据集构建的索引需要近 5T 的内存，如果考虑到为了容灾而将数据复制多份分开存储，则需要更多的存储空间，带来极大的机器成本开销。

以上是对 HNSW 算法所构建的索引的组成进行了分析，对索引所占用的存储开销及其影响因素进行了分析。下面通过实验验证 4.1.1 对内存与磁盘的访问性能差距的分析。由于是随机的读，时间上在后面读的数据无法搭前面读的数据的便车，每个读都会增加接近相同的时间。也就是说，读据花费的时间应该随着读数据次数线性递增。此外我们还将验证存放于磁盘的索引与存放于内存的索引之间存在的巨大性能差距。因此，我们使用相同的参数分别构建了两份索引，HNSW-memory 和 HNSW-disk。在搜索的时候，前者索引数据放在内存里面，而后者放在磁盘上。我们选用 ANN_SIFT1M 数据集构建了索引。构建参数设定

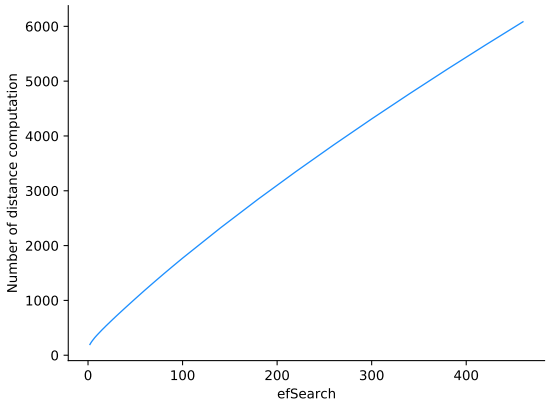


图 4-3: 距离计算次数与参数 $efSearch$ 的关系

为 $m = 16$，$efConstruction = 100$，召回数量 K 设为 10。此外，由于 HNSW 算法搜索过程中每次访问某个节点的数据都会将节点的坐标信息拿来进行距离计算，因此我们使用距离计算次数来近似搜索过程中随机访问内存的次数。最后，由于我们无法直接控制搜索时对数据的访问次数，我们是通过控制参数 $efSearch$ 来间接控制对数据的读的次数的。$efSearch$ 越大，访问的数据越多，对内存或者磁盘的访问次数也就越多，图 4-3证实了这一关系。而且我们还发现距离计算次数随着 $efSearch$ 的增大而线性递增。同时我们也看到，随着 $efSearch$ 的增大，所访问数据数量也在接近 1 万，这差不多是整个数据量的 1%。

有了以上设定，我们得到了 HNSW-memory 以及 HNSW-disk 的搜索时间与距离计算次数（数据随机读的次数）的关系，如图 4-4所示。我们看到，无论是将索引放在内存还是磁盘，单次搜索花费的时间都随着距离计算次数线性递增，证实了前面的分析。此外，我们也看到，HNSW-disk 的单次搜索花费的时间的增长趋势十分迅猛，这说明基于磁盘存放索引已经成为影响整个搜索效率的主要因素。而反观 HNSW-memory，内存访问次数的增多并没有对其搜索效率产生显著影响。搜索过程中的大部分时间还是花费在计算上。最后，HNSW-disk 的结果显示，基于磁盘的搜索效率是很低的，延迟已经超过 10ms 了，这是无法满足绝大部分实际应用的需求的。

值得注意的是，图 4-4的结果显示，基于磁盘的索引搜索时候的延时并没有达到秒级别。我分析，这里有两点原因，第一个是我们使用的是 leveldb [③]来负责将数据存放在磁盘上，而 leveldb 本身会将少部分数据缓存在内存中以加快访问

[③]https://github.com/google/leveldb

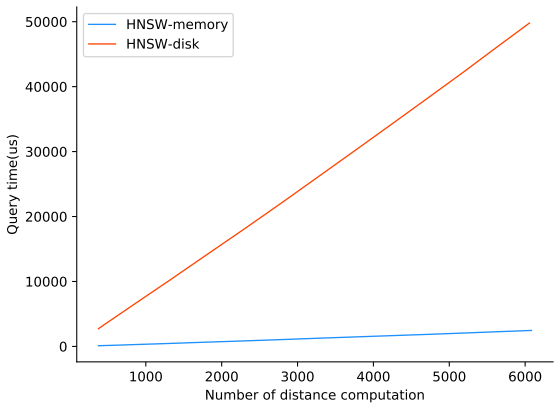


图 4-4: 单次请求花费的时间与距离计算次数的关系

速度；第二点是，本次实验环境使用的是磁盘是 SSD，而相比普通磁盘 SSD 要快数十倍。因此才有了上述结果。

通过本节的分析和实验我们知道，为了满足实际应用需求，我们需要将索引数据放在内存中，而随着数据数量和维度的升高，存放索引所需要的内存也越来越多。因此，需要采取一些策略来对索引占用的内存进行压缩，以达到降低存储索引的昂贵成本的目的。

4.2 减少索引所占用存储空间大小的方法

本节将讨论如何在 HNSW 算法的基础之上对索引大小进行有效压缩。经过 4.1对近似最近邻搜索算法所构建的索引内存占用大的问题的分析，我们知道，索引内存占用大的原因有两个，一个是需要索引的数据本身需要占用大量的存储空间，另一个是算法本身特有的附属数据结构带来的存储开销。由于这二者之间存在一定的独立性，近似最近邻算法通过一定的数据结构来组织数据，它需要用到的信息是节点与节点之间的距离，而每个节点的坐标数据就是用来计算节点之间的距离的。

基于以上事实，可以从两个方面考虑来降低构建的索引的大小。第一方向，对每个节点的坐标数据使用矢量量化或者标量量化的方法对数据进行压缩，当然这种方法是以牺牲少量准确性来换取存储空间上的极大压缩，后面我们将证明这种方式是可取的。第二个方向，针对算法本身为了索引数据而带来的附属性开销，想要直接优化会非常困难，但是可以换用其他附属性开销相对小的方

法来索引数据。

下面将分别对上述两个方向介绍。

4.2.1 使用量化方法对坐标数据进行压缩

我们知道，索引从开始构建到构建完成之后用来响应各种搜索请求，都涉及到大量节点之间的距离计算。虽然每个节点的坐标可能具有不同的实际意义，但是在近似最近邻搜索这个问题中，每个需要被索引的数据都被当作了高维空间中的一个点 (在 HNSW 中我们将每个“点”称为节点，即图上的节点)，每个节点坐标信息的唯一作用就是用来计算节点之间的距离。而量化技术就是在尽量保留节点之间的距离信息的前提下，用更少的比特去表示每个节点的坐标信息。近似最近邻搜索算法中最常用到的量化技术是矢量量化技术和标量量化技术。下面分别对二者进行简要介绍。

矢量量化是针对向量进行压缩表示的方法，由于它在压缩前后会丢失节点真实的坐标信息，因此是一种有损的压缩方法。假设我们需要对 n 个 d 维空间中的向量进行压缩表示。矢量量化采用以下方法对这些向量进行处理。对于一个 d 维的向量，假设 d 可以表示为 $d = M * D$，然后可以将 d 维空间中的向量表示为 M 个 D 维空间中向量的笛卡尔积。定义一个常数 K，我们期望对于每个 D 维的子空间，用 K 个向量来表示该 D 空间中的所有向量。具体的，对于 n 个 d 维的向量，进行维度拆分之后，每个 D 维子空间中就包含了 n 个 D 维的向量，然后从这 n 个数据中选取一部分数据（也可以是全部数据）学习 K 个聚类中心。学习完成之后，对于每个子空间都分别学习到了 K 个能够大体上代表该空间中数据分布的代表点。然后每个子空间中的 n 个子向量都用离其最近的聚类中心进行表示编码。这样对于每个子向量，它需要 $log(K)$ 个比特进行编码。就这样对所有子向量进行编码，表示每个编码后的向量需要的比特数为

$$N_{bit} = M * log(K) \tag{4-6}$$

通常情况下需要合理选取 M 和 K 的值，以使得 N_{bit} 为 8 的倍数，这样每个向量被编码后的值刚好能够被多个字节表示。

以上就是矢量量化学习（也称为训练）以及对向量进行编码的全过程。借

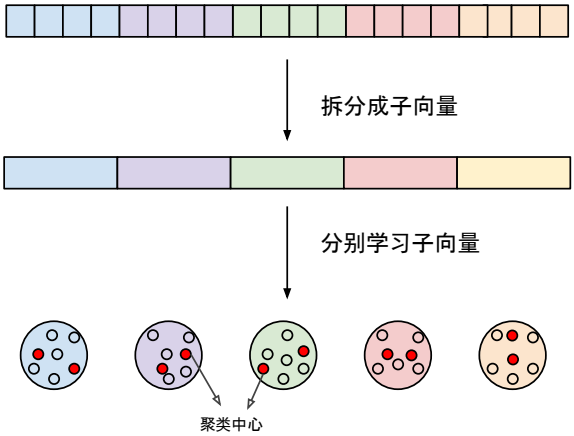


图 4-5: 矢量量化学习码本的过程。这是一个自上而下的过程，先将原始向量拆分，然后依次对拆分后的子空间学习代表点。

用通信里面的术语，学习到的聚类中心可以称其为码本，对向量进行矢量量化的过程称为编码，而将量化后的向量转换回空间中的坐标点的过程称为解码。图 4-5展示了学习码本的过程，即先将高维空间拆分成子空间，然后在子空间通过聚类方式分别学习一批聚类中心，得到码本。学习完成之后，对向量进行编码的过程和图中的学习过程也是类似的，即先对向量进行分段，只是第二过程不是学习聚类中心，而是寻找最近的聚类中心对向量进行编码。

在进行矢量量化之前，d 维空间中的点可以出现的位置是无限的，而量化后的空间能够表示的点的数量是 K^M 个。这一方面说明这种压缩是有损的，丢失了很多信息。另一方面，可以通过增大 K 和 M 来提高量化之后的空间的“容量”，进而减少信息的缺失。需要对（距离计算）损失与压缩比进行合理权衡，找到适当的折中方案。下面以实例展示矢量量化的压缩能力。

对于 n 个 1280 维的向量，假设每个维度数据用四个字节的浮点数表示。对这些向量进行矢量量化时，M 取 320，K 取 256，则每个向量压缩后需要的比特数为 $320 * log(256) = 320 * 8$ 个比特，也就是需要 320 个字节。而每个原始向量占用的存储空间为 $1280 * 4$ 个字节。这样，量化前后的压缩比为 $\frac{1280*4}{320} = 16$，即压缩了 16 倍，这一比例是相当高的。为了客观反映量化前后对召回效果的影响，后面会有实验展示压缩前后模型召回率的变化。此外，矢量量化还有额外的存储开销，$K * M$ 个 D 维的聚类中心的坐标数据需要被存储下来，有了他们才能将每个向量的编码转换为空间中的真实位置。通常情况下，K 的取值不会太大，一般取值 2^8 或者 2^{16} 就差不多了，这样由存储聚类中心带来的额外开销相对于

原始数据来说可以算作是忽略不计的，式 4-6就很好地反映了整个矢量量化过程前后对原始向量存储空间的压缩比例。

标量量化可以看作是矢量量化的特殊情况，它是针对标量进行量化而不是矢量。在使用矢量量化的时候，会将 d 维的高维向量拆分成 M 个 D 维的小向量，如果 M 与维度 d 是一样的，那么拆分出来的一维小向量实际上就是一个标量。标量量化就是针对这种特殊情况进行处理的。标量量化与矢量量化虽然在基本原理上是一致的。但是二者在技术细节以及具体实现上还是有差异的，因此在本节对其进行单独介绍。

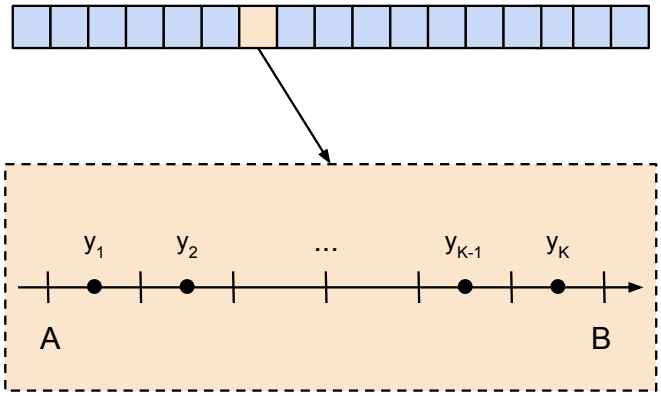


图 4-6: 标量量化的基本原理。本图自上而下地展示了标量量化的过程，对于每个维度，会选取 K 个代表点来代表数轴上的数据。这些代表点用来编码对应维度上的所有数据。

由于标量量化相对于矢量量化的粒度更细，通常情况下量化前后向量之间距离计算的误差会更小。下面简要介绍标量量化的基本原理。如图 4-6所示，标量量化是针对高维向量每个维度的标量进行的。针对每个维度的标量，它们实际上是分布在一维数轴上的一堆数。假设这堆数的分布范围为 $[A, B]$，即数据的最小值为 A，最大值为 B。进一步假设这堆数是在 $[A, B]$ 上均匀分布的，然后将该区间平均分成 K 份，第 i 份区间的中心记为 y_i。那么有了以上数据，就可以对整个区间 $[A, B]$ 上的数据进行编码，对于落在每个区间里面的数，就用区间的中心来表示这个数。这样表示之后，每个数据仅仅需要 $log(K)$ 个比特。一般 K 的取值为 2^8、2^6 以及 2^4，这样每个数据的编码需要的存储空间不到一个字节。

虽然实际上数据的分布并不严格满足均匀分布在某个固定区间中，但是这种假设实现上非常简单高效而且实际效果也很不错，因此被广泛采用。下面对标量量化的实现细节进行介绍。同矢量量化一样，需要从数据中预先“学习”一

些信息，但是标量量化需要的信息很少。只需要知道每个维度数据的最大值、最小值，而其中的每个区间的信息不需要存储，可直接通过计算获得。此外，对数据的编码和解码也变得非常简单迅速。正是因其简单、快速的特点，标量量化在实践中广泛应用。

最后用一个例子直观介绍标量量化的压缩效果。对于 n 个 1280 维的向量，假设每个维度数据用四个字节的浮点数表示。假设对每个维度的数据进行量化时，K 的取值为 256，那么向量的压缩比为 $\frac{1280*4}{1280*\frac{log(256)}{8}}=4$，即压缩了四倍。

4.2.2 采用非图结构的数据结构来组织数据

我们知道 HNSW 算法中除了高维向量本身会成为存储空间的一大开销之外，算法用于索引所有数据的图数据结构也会带来不小的存储开销。我们已经知道可以通过对高维向量进行压缩来降低原始数据的存储开销，本节我们将讨论换用其他存储开销更小的数据结构来组织数据的方法。

对于大规模的高维数据，现有方法中倒排索引是额外开销最少的数据组织方式。下面简要介绍倒排索引是如何组织数据的。首先我们知道，额外存储开销最少的组织数据的方式就是什么也不做，将所有数据连续存储，搜索的时候进行暴力搜索。当然这种方式带来的计算开销是不可接受的，但是它为我们设计更加轻量级的组织数据的方式指明了方向，即暴力搜索不需要特殊数据结构。由于每次搜索都是在局部进行的，要想加快暴力搜索的速度，需要尽快先确定搜索的大致范围，那些明显偏离目标的区域就不需要浪费时间进行遍历了。因此通过先将数据通过聚类方法进行分桶，然后搜索的时候先确定需要暴力搜索的几个桶，然后在这几个桶里面进行暴力搜索。我们用图 4-7来可视化倒排索引的基本原理。

对于倒排索引，有几个重要的参数，分别为决定对数据分成多少个桶的参数 *nlist*、每次搜索的时候需要访问的桶的个数的最大值 *nprobe* 以及最多需要访问的节点的个数 *max_codes*。参数 *max_codes* 存在的意义是在桶中数据分布不均匀的时候，如果某个桶中数据量特别多，搜索的时候不至于在某个桶中耗费过多时间。这样设计有利于平衡每次搜索花费的时间，不至于系统运行过程中出现较大的抖动。

倒排索引通常会与其他方法结合，比如我们在第 2章中介绍的，将倒排索

引与矢量量化方法进行结合的方法 IVFADC，以及进一步与 HNSW 进行结合的 IVF-HNSW 方法。需要说明的是，IVF-HNSW 才是针对大规模数据的有效方法。这是由于，倒排索引在进行搜索的时候第一步就是从 *nlist* 个聚类中心中选择 *nprobe* 个聚类中心，对于千万级数据，*nlist* 的大小是十万级的，如果这一过程暴力搜索必然是比较慢的。但是使用 HNSW 来索引这样一批数据集既不会带来很大的额外存储开销，又能有效地加快搜索速度。因此针对大规模数据，IVF-HNSW 是非常好的选择。而在 IVF-HNSW 构建的索引中，每个桶中的数据又与矢量量化进行结合，大幅较少高维向量的存储开销。

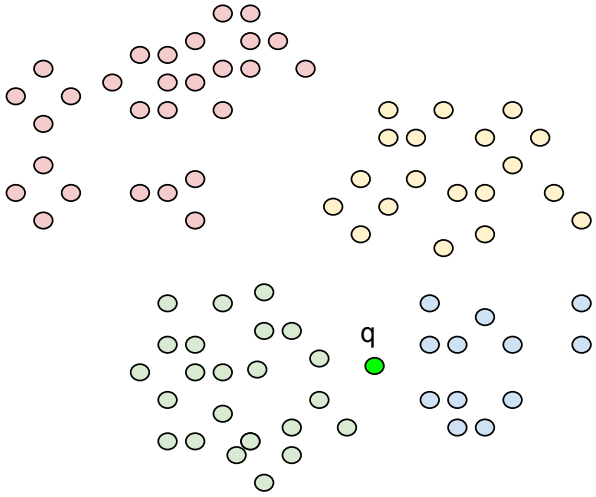


图 4-7: 利用倒排索引进行搜索。如图中所示，数据按其分布来看，大致聚集在四个区域，图中以不同颜色加以区分。对于请求 q，它离下面两个区域数据的聚类中心更近，因此搜索的时候只需要在浅绿色和浅蓝色区域进行暴力搜索就行了。

总的来说，IVF-HNSW 既减少了用 HNSW 来索引全部数据带来的额外开销，又结合了矢量量化来对高维向量进行压缩，是适合于极大规模数据的方法，而且以我目前的了解到的知识，它是最优选择之一。

4.3 对 IVF-HNSW 方法的优化

前面我们介绍了压缩近似最近邻算法所构建的索引大小的方法。通常情况下，HNSW 与标量量化结合的方法适合于对索引压缩比例要求不高但是对召回率要求较高的场景，而 IVF-HNSW 适合于对索引压缩比例要求较高但是对召回率要求不是很高的场景。目前存在的问题是，IVF-HNSW 在构建索引的速度上很慢，很难满足一些实际应用的需求。比如，某些应用每天都会产生大量的新

数据，需要及时将这些新数据构建成索引以提供服务，但是对于上千万的数据，IVF-HNSW 构建索引一般需要数小时，这就会导致新数据无法及时上线提供服务。因此，急需要采取一些措施来加速 IVF-HNSW 算法构建索引的过程。

4.3.1 对 IVF-HNSW 构建索引过程的加速

本节我们先分析 IVF-HNSW 在构建索引过程中影响构建时间的关键因素，然后提出新的方法优化构建（训练）时间。IVF-HNSW 构建索引的过程主要分为以下过程：

聚类过程。倒排索引首先需要对数据进行分桶，那么给定桶的个数的参数 $nlist$，就需要使用聚类方法 k-means 对数据进行聚类。一般来说，需要使用 IVF-HNSW 系列方法的场景都是极大规模数据的场景，这种规模下 $nlist$ 的取值是比较大的。比如，对于 1000 万条数据，通常需要 20 万个桶来分割数据，那么就需要对数据集进行 $k = 200,000$ 的 k-means 聚类。这个过程需要非常大的计算量，是非常耗时的。

编码过程。学习到了 $nlist$ 个聚类中心之后，就需要将所有数据分别添加到对应的桶中去。具体的，对于每条数据，会选择离其最近的桶，然后将数据添加到该桶中。这个过程的时间复杂度为 $O(n * nlist)$，这里 n 表示总的数据条数。可以看到该过过程也是非常快的。此外，通常情况下都会使用矢量量化对数据进行量化之后再将数据存放在桶中，由于对数据进行量化编码的过程比较快，整个编码过程都能在比较迅速的时间内完成。

根据上面的分析我们知道，影响 IVF-HNSW 构建时间的最关键因素是对数据集进行 $k = nlist$ 的聚类过程。因此只需要对该过程进行优化加速，就能大幅加快构建索引的速度。然而，天下没有免费的午餐，单纯对聚类算法进行简单替换并不能解决问题。

我们知道控制 IVF-HNSW 的搜索质量的参数是 $nprobe$ 和 max_codes，前者限定了搜索过程中搜索桶的个数的上限，后者限定了搜索过程中可以访问的数据个数的最大数量。我们需要思考为什么需要两个参数来控制搜索过程。这是因为，在构建索引的第一个阶段学习到的聚类中心并不是最优解：k-means 返回的结果是局部最优的，虽然有方法能够让聚类过程更容易得到最优解，但是会花费额外的计算量。聚类过程得到的桶的中心并不是最优的会导致最终对数据进行

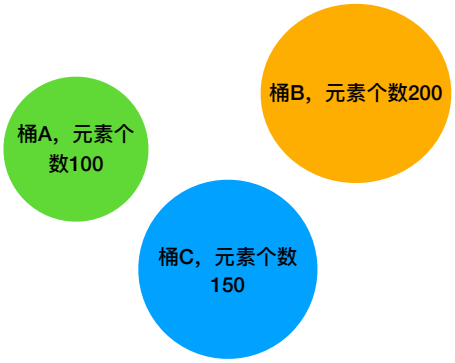


图 4-8: 对数据分桶后数据分布不均匀的情况。图中桶 A、B、C 中的数据分布不均匀，桶 A 中的数据最少，桶 B 中的数据最多。

分桶的时候数据分布不均匀。而这就是需要用两个参数 *nprobe* 和 *max_codes* 的根本原因。如果数据分布是均衡的，那么在搜索的时候只需要指定需要搜索多少个桶就行了。但是如果数据分布是不均匀的，情况就会有所不同。如图 4-8所示，桶 B 中的数据比 A 中多了一倍之多，如果搜索的时候指定只需要访问一个桶的数据就行了，那么，搜索桶 B 将比搜索桶 A 多一倍的时间。这种数据分布的不均衡导致有的搜索请求处理的更快，而有的却很慢，最终给系统带来很大的“抖动”。引入了参数 *max_codes* 之后，虽然能够让每次搜索花费的时间更加均衡，让系统更加稳定，但是也会引起召回率的下降。比如如果指定 $max\_codes = 100$，那么桶 B 的后 100 个元素以及桶 C 的后 50 个元素就永远无法被访问到，从而引起召回率的下降。

除了上去原因会引起数据分布不均衡之外，用 HNSW 算法来索引桶的中心（聚类中心）也会带来一定的误差，使得少部分数据最终并没有被分到距离它最近的桶中。

因此，如果将聚类过程中使用的 k-means 算法简单替换成速度更快的 mini-batch k-means[46] 算法，反而会加剧数据的不均衡分布问题。这是由于 mini-batch k-means 算法是以牺牲一定的精确性来换取计算速度的大幅提升。因此必须从源头来抑制数据分布不均衡的发生。我们在实践中观察到，数据分布不均衡一般具有局部性，即对数据进行分桶之后，数据只会在局部范围内分布不均衡，而不会偏离这个局部范围很远。因此，我们考虑在对数据进行分桶的时候从源头上避免数据分布极端不均衡的存在，新数据在选取被插入的桶的时候，可以有多个候选的桶。从这些候选的桶中选取既满足距离新数据近，又满足目前该桶中数据的数量并不是特别多。

算法 4.1 Balanced IVF-HNSW 的索引构建过程

function BALANCED-IVF-HNSW-BUILD
$(\Theta, n, S, nlist, efConstruction, efSearch, m, n_{candidates})$
输入:
Θ 桶不平衡系数的阈值
n 用于构建索引的数据的数量
S 用于构建索引的数据的集合
$nlist$ 对数据分桶时桶的个数
$efConstruction$ HNSW 的构建参数
$efSearch$ HNSW 的搜索参数
m HNSW 的构建参数
$n_{candidates}$ 每个数据在选择被插入的桶的时候候选桶的个数
输出:
桶的集合 I
$C \leftarrow$ 对集合 S 里面的数据采用 mini-batch k-means 方法进行 $k = nlist$ 的聚类得到的聚类中心
$HNSW_C \leftarrow$ 用集合 C 中的元素，采用 HNSW 算法，用参数 $efConstruction$ 与 m 构建的索引
将索引 $HNSW_C$ 的搜索参数设置为 $efSearch$
for $x \in C$ **do**
  $I \leftarrow I_x \cup I$ // 构建桶的集合 I
end for
for $x \in S$ **do**
  $Candidates1 \leftarrow$ 在索引 $HNSW_C$ 中搜索离 x 最近的 $n_{candidates}$ 个候选者得到的集合
  $Candidates2 \leftarrow$ 将 $Candidates1$ 中的元素按照离 x 从近到远的顺序排序得到的数组
  $target \leftarrow Candidates2[0]$ // 获得 $Candidates2$ 中的首个元素
  for $t \in Candidates2$ **do**
    $b_t \leftarrow \frac{|I_t| * nlist}{n}$ // 计算 t 对应的桶 I_t 的平衡系数
    if $b_t < \Theta$ **then**
      $target \leftarrow t$
      break
    end if
  end for
  $x \leftarrow encode(x)$ // 这里 $encode$ 泛指各种量化方法
  $I_{target} \leftarrow I_{target} \cup x$
end for
return I

基于上面的想法，我们对 IVF-HNSW 算法的构建过程进行了改进，由于我们对 IVF-HNSW 的改进主要目的是在加快索引构建的同时保持数据在桶之间分布的均衡性，我们将新提出的算法称为 Balanced IVF-HNSW，其构建过程的细节如算法 4.1所示。具体的，对数据进行分桶之后，每个桶中的数据组成了一个集合，为了简化表达，我们用 I 表示所有桶的集合。对于每个桶，我们将其定义

为桶中元素的集合，对于每个桶 $B \in I$，我们定义其负载系数为

$$B_{load} = \frac{|B|}{\frac{n}{nlist}} = \frac{nlist * |B|}{n} \tag{4-7}$$

这里 n 表示总的用于构建索引的数据的数量，$\frac{n}{nlist}$ 表示平均每个桶中的数据的数量，因此我们将桶的负载系数定义为桶的实际数据含量与平均数量之比。显然每个桶的负载系数越接近 1 越好。有了每个桶的负载系数，我们在向桶中插入数据的时候，就可以将桶的负载系数作为考虑是否将新数据插入桶中的依据。因此对于单次索引的构建过程，需要定义桶的负载系数的上限，达到这个上限的桶就不应当向其中加入更多的数据了。用们用 Θ 表示上述负载系数上限。对于 Θ 的取值，由于一般来说每个桶的数据的数量不应当超过 max_codes，即 Θ 的取值应当满足

$$\Theta < \frac{max\_codes * nlist}{n} \tag{4-8}$$

这里只是给出了一个上限，实际场景中需要根据具体情况调参。

下面给出算法 4.1的完整描述。首先在聚类过程中，会使用 mini-batch k-means 算法进行聚类。在实际实现 mini-batch k-means 的过程中，我们给算法加上了随机初始化以及提前停止迭代的机制。此外为了充分利用 CPU 的强大并行能力，我们在算法的实现上会充分将算法中能并行化的部分并行化。当然这些是实现细节，这里不再赘述。聚类过程结束之后，获得了 $nlist$ 个聚类中心，然后根据这些聚类中心将数据进行分桶。在进行数据分桶之前，会先将聚类中心采用 HNSW 进行索引，我们前面提到这种方式能够加快查找速度。随后，对于每个待插入数据，会选取一批候选的桶进行插入。首先从 HNSW 构建的聚类中心的索引中搜索 $n_{candidates}$ 个候选者桶。然后按照距离（待插入数据）从近到远的顺序依次检查每个桶，如果检查到某个桶的数据负载率未达到阈值 Θ，就将该新数据插入到该桶中，然后去处理下一个数据。通常情况下，数据被插入桶之前，都会经过矢量量化等量化方法对数据进行压缩编码，这里用 $encode$ 统一表示这一过程。

可以发现，以上构建索引的过程同原来的构建过程的核心差异是我们承认了分桶偏差的存在。这种偏差来自于聚类过程以及 HNSW 搜素最近邻的过程。前者是聚类过程陷入局部最优引起的，后者是近似最近邻算法天然存在的。在

承认这种偏差存在的必要性之后，我们通过调整数据在局部范围内的分布，来提高所有数据分桶的均衡性，进而提高算法稳定性。

本节提出对 IVF-HNSW 算法的改进的核心目的是加快构建/训练速度。这一核心改进是通过 mini-batch k-means 算法对 k-means 的加速来完成的。前者每个迭代过程并不使用全部数据，而是从中随机采样一个 *batchsize* 这么大数量的数据来进行迭代。由于是随机采样，采样的数据的分布是和原数据基本一致的，因而采样的数据也能帮助我们学到数据集的聚类中心。一般来说，*batchsize* 的大小会远小于原始数据数量的大小，因而采用 mini-batch k-means 能够大幅减少计算量，加快索引构建（训练）过程。

4.4 实验与分析

本节分别就 HNSW 与 IVF-HNSW 展开实验。首先展开的是 HNSW 与量化方法进行结合的实验，接着进行 IVF-HNSW 以及 Balanced IVF-HNSW 的实验。实验环境都是统一的，采用的是 Intel i5 处理器，包含 48 个 CPU 核以及 128GiB 内存。这么大的内存保证了能够容纳下实验过程中构建的索引。此外，实验数据采用的是实际生产中的应用数据，它们是不公开的。

4.4.1 HNSW 与量化方法进行结合的实验

本节将展示将 HNSW 与量化方法结合后对存储空间的压缩效果。我们对比了三个方法，分别是不采用量化方法的 HNSW 算法、采用标量量化的 HNSW 算法以及采用矢量量化的 HNSW 算法，我们分别记为 HNSW、HNSW_SQx 以及 HNSW_PQy。这里 x 和 y 分别为一个数字，前者表示标量量化后每个标量使用多少个比特进行表示，后者表示矢量量化时候的参数 M 的取值，即子向量的个数。此外，矢量量化的参数 K 固定为 256，即每个子向量量化后用 8 个比特（1 个字节）进行表示。

由于向量本身和近似最近邻算法之间的低耦合性，我们很容易将近似最近邻算法同量化方法进行结，这里就无需赘述 HNSW 与量化方法的实现细节。值得注意的是，量化方法有学习、编码以及解码过程。这些都是计算比较密集的过程，我们采用了成熟的开源实现进行实验。在成熟的开源近似搜索算法中，faiss

是将 HNSW 与量化方法结合实现最好的。因此本论文中矢量量化与标量量化就是采用的 faiss 的实现。我们分别在三个非公开的数据集上进行了实验。这三个数据集都是实际应用中的真实数据集，由于这三个数据集是与视频相关应用的数据集，我们分别称这三个数据集为 VIDEO-01、VIDEO-02 以及 VIDEO-03。

VIDEO-01 以及 VIDEO-02 都是维度居中，但是数据量较大的数据集，这种数据集只要数据量达到稍大规模，所构建的索引所占用存储空间大小就会变得比较大。针对这种情况，我们决定采用标量量化，达到对向量存储空间压缩四倍的效果。数据集 VIDEO-01 以及 VIDEO-02 的详情以及在这两个数据集上标量量化的表现分别见表格 4-2和表格 4-3。

表 4-2: HNSW_SQ 在数据集 VIDEO-01 上的实验

数据信息	数据名称	距离	数据维度/ 每个维度数据类型	数据数量 （万）	K	数据大小 （GiB）
	VIDEO-01	欧式距离	512/4 字节浮点数	1,000	500	19.7

实验结果	方法	构建参数	搜索参数	训练时间 （min）	召回率 (%)	平均召回时间 （ms）	索引大小 （GiB）
	HNSW	$m = 32$, $efConstruction = 500$	$efSearch = 500$	58	95.1	4.8	22.3
	HNSW_SQ8	$m = 32$, $efConstruction = 500$	$efSearch = 500$	32	94.9	4.8	**7.6**

表 4-3: HNSW_SQ 在数据集 VIDEO-02 上的实验

数据信息	数据名称	距离	数据维度/ 每个维度数据类型	数据数量 （万）	K	数据大小 （GiB）
	VIDEO-02	欧式距离	768/4 字节浮点数	500	1,000	14.3

实验结果	方法	构建参数	搜索参数	训练时间 （min）	召回率 (%)	平均召回时间 （ms）	索引大小 （GiB）
	HNSW	$m = 16$, $efConstruction = 500$	$efSearch = 1000$	23	96.5	9.8	15.0
	HNSW_SQ8	$m = 16$, $efConstruction = 500$	$efSearch = 1000$	23	94.6	11.0	**4.3**

表格 4-2显示，1000 万 512 维的原始数据的存储空间就需要 19.7GiB 的内存空间，占用的内存空间还是比较大的。一般来说单个计算机可能无法存下这么多的数据。HNSW 构建完的索引占用内存大小为 22.3GiB，这说明 HNSW 算法固有附属性数据结构占用的存储空间为 2.5GiB 左右，这个存储开销相对于原始向量来说影响较小。采用 HNSW_SQ8 进行压缩的索引最终大小为 7.6GiB，减去索引的附属性数据结构 2.5GiB 之后，剩下的压缩后的向量占用存储空间为 5.1GiB，

而这一结果和原始向量的四分之一是很接近的。而理论上原始向量的压缩比为 $\frac{512*4}{512*1}=4$，这个和实际观测结果是几乎一致的。最终压缩后索引大小为 7.6GiB，这个大小单个计算机就能够完全存储下来，可见压缩效果是非常好的。召回率和平均召回时间衡量了最终构建完成的索引“质量”。我们看到，HNSW_SQ8 在对索引大小压缩四倍的情况下，保证了搜索时间基本不变且召回率只有略微的下降。这个效果是非常喜人的。因为压缩之后的模型召回率依然能够达到 0.95 左右，而这个是许多不采用量化方法的传统算法都难以达到的。

表格 4-3进一步证实了标量量化的强大能力。这里能力指的是能够减少量化前后向量之间距离的误差的能力。在该 768 维数据集上，HNSW_SQ8 同样达到了对索引大小进行大幅压缩的效果的情况下几乎保持了召回率和召回时间不变。

还有一个没有被讨论的观测指标是训练时间。训练时间既包含标量量化的学习时间也包含对所有向量进行编码然后构建索引的时间。我们观察到在 512 维的数据集上，HNSW_SQ8 的训练时间更快。这是因为，512 这个向量维度是 2 的整数幂，在计算的时候能够利用到计算机针对这种与 2 的整数幂对齐的数据的专用优化。所以训练时间能够更快，当然这种加速并不是绝对的同时也不在本章的主要讨论主题内，因此这里只是简要提及。

由于矢量量化一般适用于进行比较大的幅度的向量压缩，比如前面提到对 1280 维的数据进行高达 16 倍的压缩。下面我们采用维度为 1280 维的数据集 VIDEO-03 来观测矢量量化的效果。如表格 4-4所示，HNSW_SQ8 依然保持了非常好的效果。HNSW_PQ320 虽然达到了大幅压缩索引的效果，但是其召回率只有 0.8 左右。虽然这个召回率已经能够满足一些应用的需求，但是还是不尽人意，一般来说，召回率达到 0.9 就能够满足大部分应用的需求了。此外，我们发现 HNSW_SQ8 以及 HNSW_PQ320 在召回速度上相对 HNSW 有所提升，我们认为这个和实现相关，涉及到各种并行化加速方法，这里不对此进行讨论，因为我们主要关注的是模型压缩后召回率的情况。

针对 HNSW_PQ320 在对向量进行大幅进行压缩后出现召回率大幅下降的问题。这里给出分析，我们认为主要原因是矢量量化对向量精度的损失较大。我们用图 4-9来解释这一损失产生的原因。在图中，如果采用原始距离向量计算节点之间的距离，那么节点 A 就是离请求 Q 最近的节点。但是将节点进行量化之后，A，B，C 三个节点量化后的节点离请求 Q 的距离是十分接近的，这就导致

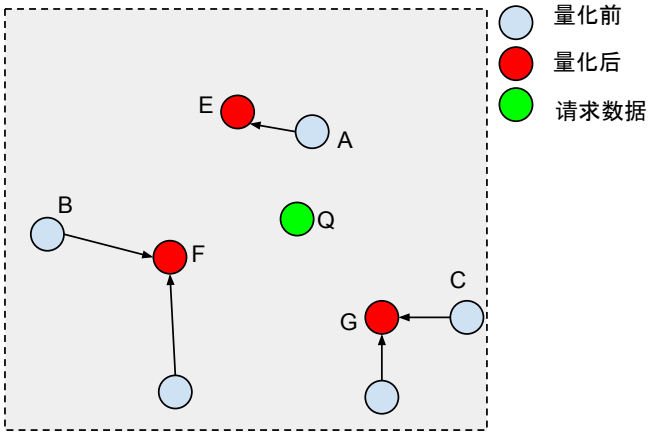


图 4-9: 矢量量化的距离损失。对一批数据进行矢量量化就选用一批点来代表这批数据，图中用二维平面展示了量化前后节点之间距离计算的差异。在对节点进行量化之后，请求数据 Q 与数据 A、B、C 之间的距离变成了数据 Q 与这些数据被量化之后的节点 E、F、G 之间的距离。

了最终返回离 Q 最近的节点并不是节点 Q。此外，由于每个节点在量化之后，它的位置就会发生偏移，导致访问的节点越多，积累的误差也就越多。如果能够将搜索固定在某个局部范围之内，那么就能够较少遇见位置出现偏差的“错误”节点。但是 HNSW 算法无法满足这一要求，HNSW 的搜索过程总是全局的，而且 HNSW 尝试通过访问尽可能多的节点来提高召回率。我们下面将介绍，基于倒排索引可以解决这个问题。

4.4.2 对 IVF-HNSW 以及 Balanced IVF-HNSW 的实验

为了清楚表达 IVF-HNSW 使用的矢量量化方式，我们采用 IVF-HNSW_PQx 来表示构建的模型名，其中 x 表示进行矢量量化的时候参数 M 的取值。

表格 4-4的结果显示，同样是对原始向量进行了 16 倍的压缩，方法 IVF-HNSW_PQ320 构建的索引相对于 HNSW_PQ320 来说更小，这和我们前面的分析是一致的，即倒排索引相对于 HNSW 算法只需要更少的数据结构来组织所有数据。在另一项重要指标召回率上，IVF-HNSW_PQ320 也达到了很好的水平，在整体内存占用量压缩 10 多倍的情况下还能够保持非常高的召回率。之所以 IVF-HNSW_PQ320 能够在召回率上领先 HNSW_PQ320 很大幅度，是因为 IVF-HNSW 系列方法成功地将最终搜索的范围锁定到了局部的范围之内，而避免了 HNSW 搜索过程中由矢量量化引起的全局误差。由于 HNSW 算法的搜索范围是

全局数据，搜索过程中访问过的所以节点都有可能影响最终的结果，而矢量量化带来的误差导致，访问的节点的范围越大，被错误当作最近邻的节点也就越多，因而带来了全局误差。

表 4-4: HNSW_SQ、HNSW_PQ 以及 IVF-HNSW_PQ 在数据集 VIDEO-03 上的实验

数据信息	数据名称	距离	数据维度/ 每个维度数据类型	数据数量 （万）	K	数据大小 （GiB）
	VIDEO-03	欧式距离	1280/4 字节浮点数	149	1,000	7.3

实验结果	方法	构建参数	搜索参数	训练时间 （min）	召回率 (%)	平均召回时间 （ms）	索引大小 （GiB）
	HNSW	$m = 32$, $efConstruction = 460$	$efSearch = 1,000$	15.1	94.5	1.6	7.5
	HNSW_SQ8	$m = 32$, $efConstruction = 460$	$efSearch = 1,000$	7.5	94.0	0.9	2.2
	HNSW_PQ320	$m = 32$, $efConstruction = 460$	$efSearch = 1,000$	12.0	81.5	0.7	0.84
	IVF-HNSW_PQ320	$m = 32$, $efConstruction = 460$, $nlist = 10,000$	$nprobe = 128$, $max\_codes = 100,000$	55.4	90.0	1.7	**0.61**

从召回率和内存压缩上的表现来看，IVF-HNSW 是非常优秀的。但是我们发现 IVF-HNSW 的训练时间相对于其他方法来说长了很多。这也是我们前面提到的问题，下面来看 Balanced IVF-HNSW 对索引构建的加速效果。

表 4-5: IVF-HNSW 与 Balanced IVF-HNSW 在数据集 TEXT-01 上的实验

数据信息	数据名称	距离	数据维度/ 每个维度数据类型	数据数量 （万）	K	数据大小 （GiB）
	TEXT-01	欧式距离	128/4 字节浮点数	1,100	1,000	5.7

实验结果	方法	构建参数	搜索参数	训练时间 （min）	召回率 (%)	平均召回时间 （ms）	索引大小 （GiB）
	IVF-HNSW_PQ16	$m = 32$, $efConstruction = 460$, $nlist = 262,144$, $niter = 3$	$efSearch = 1,000$, $nprobe = 1,000$, $max\_codes = 10,000$	176.8	**94.7**	6.7	0.47
	Balanced IVF-HNSW_PQ16	$m = 32$, $efConstruction = 460$, $nlist = 262,144$, $niter = 3, batchsize = 10,000$	$efSearch = 1,000$, $nprobe = 1,000$, $max\_codes = 10,000$	**36.0**	90.1	**6.3**	**0.47**

对 Balanced IVF-HNSW 采用了两个实际生产中用到的数据集。它们是从文本相关应用中获得的数据，我们分别称之为 TEXT-01 以及 TEXT-02。前者是包含 1100 万条 128 维的数据，后者包含 1300 万 128 维的数据。在这两个数据集上，我们设定参数 $nlist$ 的大小为 20 万左右，因此平均每个桶的数据是 500 左右，而我们将搜索参数 max_codes 设定为 1 万条数据，因此我们将每个桶的负载系数上限 Θ 设定为了 10，这样每个桶的最大数据数量为 5000，接近 max_codes 的

表 4-6: HNSW 与 IVF-HNSW 在数据集 TEXT-02 上的实验

数据信息	数据名称	距离	数据维度/ 每个维度数据类型	数据数量 （万）	K	数据大小 （GiB）
	TEXT-02	欧式距离	128/4 字节浮点数	1,300	1,000	6.2

实验结果	方法	构建参数	搜索参数	训练时间 （min）	召回率 (%)	平均召回时间 （ms）	索引大小 （GiB）
	HNSW	$m = 50$, $efConstruction = 2000$	$efSearch = 3,000$	62.1	**93.1**	**4.3**	12.0
	Balanced IVF-HNSW_PQ16	$m = 32$, $efConstruction = 460$, $nlist = 262,144$, $niter = 3, batchsize = 10,000$	$efSearch = 1,000$, $nprobe = 1,000$, $max\_codes = 10,000$	**36.8**	90.7	5.5	**0.50**

一半。此外，我们将 k-means 和 mini-batch k-means 的迭代次数 $niter$ 都设定为 3。值得注意的是，在 mini-batch k-means 算法中，$niter$ 表示在整个数据集上迭代的轮次，在确定好每次迭代的 $batchsize$ 大小之后，迭代次数为

$$iter_{batch} = \frac{niter * n}{batchsize} \tag{4-9}$$

由于对 mini-batch k-means 采用了提前停止的机制，因此算法一般不会训练完全部的轮次。提前停止的机制设定为如果一定的迭代轮次之后，整体错误率还不下降，则停止训练。

首先展开的是 IVF-HNSW 与 Balanced IVF-HNSW 的对比实验。实验结果如表格 4-5 所示。首先关注训练时间，方法 Balanced IVF-HNSW 训练时间是 36 分钟，而原始算法训练时间多达 176 分钟，接近三个小时。从这个角度看，本文对 IVF-HNSW 的改进顺利实现了大幅加快索引构建的目的。量化方法均是采用参数 $M = 16$ 的矢量量化，量化前后向量压缩比为 $\frac{128*4}{16*1} = 32$ 倍。这也从表格中的最终索引大小反映出来了，最终索引大小不足 0.5GiB。对于千万级的数据，这是很大的压缩比。而最终模型的效果，在平均召回时间上，Balanced IVF-HNSW 与 IVF-HNSW 基本持平，而在召回率上前者出现了小幅下降，这是因为使用 mini-batch k-means 导致在最终聚类效果上相对 k-means 算法有所下降。但是最终的召回率 0.901 依然维持在较高水准，符合绝大多数应用需求。总体来说，以小幅的召回率的牺牲换取训练时间的大幅减少，是符合实际应用需求的。

下一个实验中在 TEXT-02 上对比了 HNSW 与 Balanced IVF-HNSW，前者是目前综合表现最好的算法之一但是未采用任何量化方法，后者为了较少内存占用结合了量化方法与倒排索引，可以说已经将索引模型压缩做到了极致。实验

结果如表格 4–6所示。Balanced IVF-HNSW 同样使用了向量压缩倍数为 32 的矢量量化。最终 Balanced IVF-HNSW 构建的索引大小只有 0.5GiB，而 HNSW 构建的索引大小多达 12GiB，足足是前者的 20 多倍。此外，HNSW 在构建索引的时候引入的额外存储开销占比达到 50% 左右，这是因为此时数据量已经比较大了，HNSW 构建参数 m 的取值也就相应比较大，导致存储图结构的开销变大。在召回时间上，HNSW 略微占有，但是 IVF-HNSW 的结果也很不错。在召回率上，HNSW 要高出 3 个百分点，但是 IVF-HNSW 的结果也达到了很优秀的水平。最后在训练时间上，IVF-HNSW 还要快于 HNSW，这是因为 HNSW 对于每个新数据的插入都需要进行复杂的搜索过程，数据越多 HNSW 构建也就越慢。总体来说，IVF-HNSW 经过本文提出的算法 4.1的优化，在大幅压缩索引模型大小的情况下，其他各项指标都达到了比较优秀的水平。是在内存资源比较紧张的情况下的优先选择。

本节展示了 IVF-HNSW 与矢量量化结合之后对索引的强大压缩效果，同时也验证了 Balanced IVF-HNSW 对 IVF-HNSW 的大幅加速效果。在大规模数据集上，Balanced IVF-HNSW 能够满足训练速度快，索引压缩比例大同时保持较高召回率，是这种场景下的重要选择之一。但是，在实践中也发现，Balanced IVF-HNSW 存在过多的参数需要人为设定，带来很大的学习成本以及增加了调参时间成本，这也限制了它的广泛应用。

4.5 本章小结

针对近似最近邻搜索算法存在的内存占用大的问题。本章首先以 HNSW 为例分析了近似最近邻搜索算法所构建的索引的组成。然后论证了为什么索引数据必须存放在内存中。接着从两方面提出了解决方案，并分别通过一系列分析与实验，验证了各个解决方案的效果。

一方面，HNSW 构建的索引很大一部分组成成分是原始高维向量，因此可以将矢量量化/标量量化同 HNSW 等近似最近邻搜索算法结合，压缩索引大小。另一方面，HNSW 算法本身用来索引数据的图数据结构会带来比较大的存储开销，可以使用倒排索引这种更加轻量级的结构来组织数据，以减少附属数据结构的开销。在倒排索引的框架之内，用 HNSW 来索引聚类中心，然后使用矢量量

化对高维向量进行压缩的方法 IVF-HNSW 是非常适合于大规模数据的方法，但是它存在索引构建速度慢的问题，限制了它的广泛应用。针对这个问题，本文提出了新的索引构建方法 Balanced IVF-HNSW 来加速 IVF-HNSW 的构建过程，让 IVF-HNSW 变得更加实用化。

经过本章以及第 3章的分析，这里总结了以下选取合适的近似最近邻搜索算法的准则。首先，如果对召回率要求极高，且数据规模不是很大（一般来说不到千万级这个规模的数据可以认为规模不大），且有足够的 GPU 计算资源，可以选用线性扫描。如果没有 GPU 资源，优先选用 HNSW。如果内存资源比较紧张，可以考虑采用 HNSW 与标量量化的结合方法 HNSW_SQx。如果数据规模非常大且内存资源十分有限，需要对索引大小进行大幅度压缩，考虑采用 Balanced IVF-HNSW 方法。

第五章　应用：微信分布式特征检索组件 SimSvr

前面章节对具体算法如 HSNW，IVF-HNSW 等进行了分析与优化，然而这些算法离被应用到生产环境中还有一段距离。本章将基于我们的应用实例，微信工业级近似最近邻搜索组件 SimSvr，介绍工业级应用场景中对近似最近邻搜索组件的实际需求是怎样的，接着本章将介绍 SimSvr 的系统架构与功能特性，以及我们熟知的 HNSW、IVF-HNSW 等算法在该系统中处于怎样的位置。最后本章将介绍目前 SimSvr 的实际应用情况。

5.1　应用背景

互联网的发展带来了内容的繁荣，也孕育出了各种搜索推荐系统。这些系统对 K-ANNS 的需求十分强烈。以微信的场景为例，在看一看、搜一搜等应用中，存在大量的文本、图片、视频指纹以及其他各类数据的检索需求。在这些应用场景中，一般都要求检索系统能够支持亿级以上规模的数据，而且系统需要至少满足稳定、高性能、可扩展这三点特性。为了保证支持亿级以上规模的数据，同时保持可扩展性，需要采用分布式的框架，因为单机无法容纳这么大的数据且无法扩展。此外，单机也无法保持稳定性，因为单机故障会直接导致整个搜索组件无法提供服务。而开源的成熟 K-ANNS 库如 hnswlib，faiss 等都是针对单机而设计的，它们虽然在单台机器上具有很高的性能，但还是无法应对上述需求。

根据以上讨论，我们知道一个分布式的近似最近邻搜索组件才能满足我们实际场景的需求，下面对这个组件需要满足的要求进行细化：

高召回率。这是近似最近邻搜索算法能够满足所有应用基本需求的前提。在第 3章和第 4章中我们介绍的 HNSW、IVF-HNSW 以及 Balanced IVF-HNSW 都是在任何时候都能保证高召回率的算法，甚至有的时候为了极高的召回率会采用线性搜索。正是由于对召回率的高要求，决定一个近似最近邻搜索组件好坏

的关键因素就是其采用的近似最近邻搜索算法。因此，在一个近似最近邻搜索组件中，一般将其中的近似最近邻搜索算法部分被称为搜索引擎。

高性能。这里性能的主要含义是在搜索延时上的表现，当然搜索延时越低，每秒能够完成的请求数（Query Per Second，简称 QPS）也就越多，因此高性能也可以理解为高 QPS。一般来说，即使是检索一个上亿规模的数据集，单次搜索的延时也不应当超过 10ms。这是因为对于许多在线上提供服务的应用来说，近似最近邻搜索只是其服务中的一环，比如某些应用会将召回的结果进行更进一步的处理，此外由于网络也有一定的延迟，这些延迟加在一起如果过大，会非常影响用户体验。有些搜索引擎，比如 faiss，会将近期搜索请求组织在一起进行批量处理。这种方式虽然会增大单个搜索请求的响应时间，但是能够提高整体的吞吐量，在满足单个搜索请求延时要求的前提下，批处理也不失为一种很好的请求处理方式。

稳定性。稳定性可以理解为随时随地都可以获得服务。也就是对于建好的索引，能够随时随地响应请求，并且能够返回期望的结果。这一方面必然要求作为搜索引擎的核心算法能够满足高召回和高性能的条件。另一方面，由于是在分布式的环境中，如果某台机器出现了故障，也要保证服务的可靠性。这就涉及到在分布式系统中如何保持服务的可靠性的问题了。由于我对分布式理论不是很熟悉，这里只做简单介绍。首先，由于单个机器可能发生故障等情况，如果某个机器上的由于出现故障导致某个索引的搜索服务不可用，应该能有其他未故障的机器能够接替它的工作。这就需要构建的索引能够复制多份，分布在不同的机器上，而且最好不处于同一个机房。如果某些应用对 QPS 要求高，那么将单个索引复制多份并分布在不同的机器上就能够线性叠加 QPS，当然这里需要做好负载均衡的处理，让请求均匀分布在单个索引的过个副本上面。此外，对于一些重要的数据如构建的索引数据还需要保证它们不会丢失。

可扩展性。可扩展性指的是，随着时间的迁移，系统可能需要增加新的索引或者删除某些索引，以及可能需要对某些索引进行动态调整等等，这些变化都不需要对系统进行过多的改动而系统依然保持活力，能够正常提供服务。比如需要增加新的索引的时候，只需要增加新的机器来容纳新的数据而不需要进行过多其他的修改。

动态更新。动态更新指的是某个索引数据的动态更新，在概念上它应当算

作系统可扩展性的一部分，但是由于它是在近似最近邻搜索应用中的常见需求，所以在这里特殊说明下。许多应用，每时每刻都有源源不断的新数据产生，比如互联网中的图片搜索引擎，每天都需要针对新产生的图片数据建索引。这里就产生一个问题，新的数据如何添加到现有索引中去。是在旧索引上直接添加新的数据还是将所有数据整合在一起再构建新的索引。索引构建过程是否会影响正在提供的服务。更新的数据需要多久时间才能被检索到。这些问题都是动态更新需要考虑的。

其他功能特性。上面讨论的都是一个分布式近似最近邻搜索框架的核心功能，它们需要在系统设计的时候充分考虑各种因素，合理选取各种技术实现。这里介绍的其他功能对系统设计上难度不大，但是对于一个近似最近邻搜索组件的使用者来说是有用的功能。第一点是支持多种不同的索引，比如一个检索系统能够同时容纳图片搜索索引又能容纳文本搜索索引且二者互不干扰。第二点是支持数据的版本控制，比如每一份索引数据都包含不同版本的数据，能够根据版本号检索不同版本的数据。第三点就是需要支持搜索结果的过滤等操作，比如期望搜索结果不包含位于给定黑名单里面的数据。这里需要注意的是，如何在过滤结果的同时保证返回给定数量的的结果。这里介绍了一些常见功能特性，可能还有其他需求，这里不再赘述。

有了上述对分布式近似最近邻搜索组件的需求，接下来将介绍微信的工业级近似最近邻搜索组件 SimSvr，看它是如何满足上述对系统的要求的。

5.2 SimSvr 总体设计

本节将介绍 SimSvr 的总体设计。包括索引数据的组织方式、系统的总体架构以及关键技术分析这三个方面。下面分别就每个方面展开分析。

5.2.1 索引数据组织方式

在设计索引数据的组织方式的时候有两个组织原则。第一个原则是一份索引数据要拷贝多份，然后分布在不同的机器上，以提高系统冗余和极限 QPS。第二个原则是每一份索引数据不能太大，否则单个机器的内存无法容纳下这么大的索引数据。综合这两个因素，SimSvr 采用了先对数据分块然后再复制多份的

思路。通常我们将一个运行中的 SimSvr 组件称为一个 SimSvr 实例（或简称实例），对于一个实例，它被设计为可以容纳多个不同种类的索引，SimSvr 中用 table 表示一种索引。比如有两份不同的数据都要构建索引，那么就需要构建两个 table。对于每个 table，如果数据量过大，需要对数据先进行拆分然后构建索引，拆分之后的每份数据被称为一个 shard，对于一个 shard 的数据，其构建的索引会被复制多份，其中每一份索引被称为一个 part。

综合上面的信息，SimSvr 使用 tx_sy_pz 表示 table x 中第 y 个 shard 构建的索引的第 z 个副本，这样一份索引被称为一个 container。一般 x、y、z 这三个数字自 0 开始编号，比如 t0_s0_p0，t0_s0_p1 分别表示第一个 table 的第一个 shard 的第一个 part 和第二个 part。

container 是对索引数据进行调度的基本单位。一般来说，对于每个 shard 构建的多个 part 的索引，每个 part 需要分布在不同的机器上，这样才能提供冗余并提高并发量。同一个 table 的不同 shard 的索引在存储上并没有相关关系，但是对于每次搜索，单个 table 的每个 shard 都会被访问到，然后综合几个 shard 的结果得到最终的结果。

5.2.2 系统架构

SimSvr 的系统架构如图 5-1所示。我们将其主要分为四个部分，第一个部分是 master，它负责整体的调度；第二部分是持久信息存储部分，主要包括 Data FS，Chubby 以及 Index FS，他们分别负责原始数据、路由信息与各种元信息以及构建完成的索引的持久化存储；第三部分是 trainer，它负责根据 master 的指示去拉取 Data FS 中的原始数据来构建索引然后将构建完成的索引存储到 Index FS 中；第四部分是 worker，它是负责提供在线检索服务的，其中的每一台机器都容纳了一系列的 container。

Data FS、Chubby 以及 Index FS 是 SimSvr 的外部依赖，它们的作用分别是：

Data FS。全称为 Data filesystem，它是一个分布式的文件系统，使用 SimSvr 的用户会将需要构建索引的原始高维向量放在这里。当 master 检测到这里的数据有更新之后会拉取这里的数据取构建索引。

Chubby。是一个提供分布式锁服务的系统。它存储元数据（如 Data FS 中数据文件的目录）、路由信息以及 worker 的状态信息等。这里路由信息用于指导

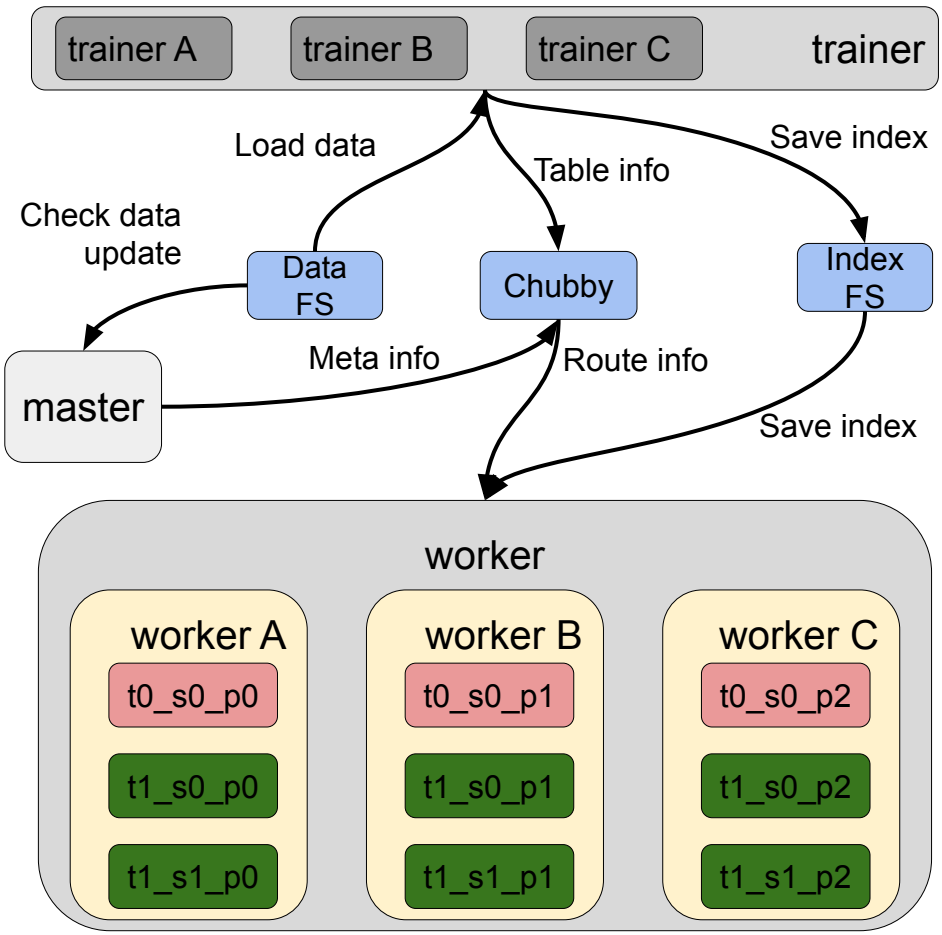


图 5-1: SimSvr 系统架构。图中箭头方向表示主要信息的流动方向。

访问某个 container 中的索引需要访问哪台机器。由于这些信息都是随时变动的，Chubby 保证了这些信息的分布式一致性。

Index FS。全称为 Index filesystem，同样是一个分布式的文件系统。负责持久化构建完成的索引，在需要的时候 worker 会将索引读进内存来提供服务。

总的来说，Data FS 与 Index FS 负责数据与索引的存储，并通过底层的分布式架构保证数据的安全性。而 Chubby 保存着系统运行时的各种状态信息，同时保证这些信息在分布式环境中的一致性。

master 是整个 SimSvr 的核心，它监控者系统的实时状态并做出各种决策，它包含一系列功能。（1）数据调度。通过综合 table 的元信息以及 worker 的状态，将未分配 table 数据分配到 worker 上面或者将失效的 worker 上的数据调度到有效的 worker 上面。table 的元信息是在索引构建完成之后保存在 Chubby 上的。（2）生成集群路由表。根据各个 worker 的负载情况等状态信息生成集群的路由表。client 端（即用户请求端）首次发起请求的时候会随机向一位 worker 服

务器发起请求，该 worker 会返回路由表。client 将路由表进行缓存，下次请求就可以根据路由表的信息向特定 worker 发起检索请求。（3）感知数据更新。当检测到 Data FS 中的数据发生了更新，就会新建一个任务让 trainer 拉取数据构建索引。master 是一个无状态的服务。数据调度以及路由信息的分布式一致性是通过 Chubby 来保证的。

tainer 负责索引的构建。trainer 被设计为每次都只构建一个 shard 的数据，对于一个 table 的多个 shard 可以通过叠加 trainer 的数量来加快构建索引的速度。这样设计是因为单机往往无法容纳下整个 table 的数据。

worker 是整个 SimSvr 组件中负责向外界提供服务的部分，它的功能如下。将索引加载到内存中，master 将每个 worker 应当持有的索引信息更新到 Chubby 中，worker 通过轮询 Chubby 来进行索引的加载与卸载。对 client 提供检索服务，所有的 client 的检索请求都是通过 worker 来完成的。

5.2.3 关键技术

下面介绍系统设计中的一些关键技术，包括搜索引擎的选择，数据调度以及数据更新。

搜索引擎。我们已经知道 trainer 负责索引的构建，worker 负责加载索引提供搜索服务，它们使用的核心算法被称为搜索引擎。SimSvr 一开始选择的搜索引擎是 HNSW，这是因为 HNSW 是目前综合表现最好的算法之一。SimSvr 采用了 HNSW 的官方实现 hnswlib，而本论文作者是 hnswlib 库的第二大贡献者。后续为了解决构建的索引的内存占用大的问题，引入了 IVF-HNSW 方法，借鉴了开源实现 faiss。本论文作者负责将 faiss 接入 SimSvr 并对 IVF-HNSW 进行了一定的优化，第 4章中的 Balanced IVF-HNSW 就是在 faiss 实现的基础上进一步修改然后整合进 SimSvr 的。

总的来说，搜索引擎作为 SimSvr 的核心部分，本论文作者对 SimSvr 的主要贡献也主要体现在搜索引擎上面。

数据调度。索引以 container 为基本单位被 master 所调度。前面已经介绍了 worker 会轮询 master 更新到 Chubby 上的 container 调度信息。在系统运行过程中，主要存在两种类型的数据调度。第一种是某个机器因为故障导致暂时无法提供服务，监控到某个 worker 故障之后，master 会及时将受影响的 container 调

度到其他机器上。第二种类型是当某个 table 目前的索引副本数量无法满足服务的需求，需要扩增副本，master 会将扩增的副本调度到有空闲空间的机器上。数据调度完成后，各种路由信息，元信息也会被及时更新，以保证 client 端能访问到合适的 worker。

数据更新。许多任务对于构建的索引都有更新数据的需求，SimSvr 的架构被设计为读写分离的架构，不适合对数据的实时更新。这里读写分离指的是，索引的读和写是分离的。索引的写过程，即构建过程是由 trainer 负责的，索引的读过程是 worker 将索引加载到内存中提供搜索服务的过程。这两个过程是由不同的任务负责的，且只有当 trainer 构建好索引才能供 worker 进行加载使用。这种方式能够保证 worker 提供的在线检索服务不受数据更新的影响。

SimSvr 之所以采用读写分离的架构，是因为在分布式架构中，如果让 worker 支持分布式地构建索引，为了保证数据一致性会需要进行精心复杂的设计。而读写分离的架构中，每个 container 的构建是由单个机器负责的，不需要考虑分布式一致性。因此，SimSvr 对数据更新的支持采用的方案是仍然让 trainer 先构建，worker 再加载的模式。

由于这种模式下更新的数据无法及时反映在 worker 中，SimSvr 对于很小规模（数千条数据）的数据更新采用了另外的策略。具体的，对于小批量的数据，会先用分布式文件系统将数据存储下来，然后让 worker 读取这些数据把这些数据更新到现有的索引中。在插入到现有的索引的过程中，会使用很少的 CPU 资源，以尽量保证不影响线上服务。这里采用分布式文件系统预先存储数据的方式是为了保证即使本次更新失败，数据不会丢失。这样如果 worker 在更新索引的时候发生了失败，它可以从接着从 Index FS 加载原始索引，然后接着加载需要更新的数据继续之前失败的索引构建过程。

5.3 服务现状

根据 SimSvr 的技术总结文章[①]的介绍，截至 2020 年 7 月，SimSvr 已经部署了 160 多个模型，总索引的数据量已经超过 20 亿，广泛应用于微信视频号、看一看、搜一搜等业务。

[①]https://mp.weixin.qq.com/s/rXXm6c8LrTqqP4iWf9mtxA

以 SimSvr 在搜一搜上面的应用为例。搜一搜采用 SimSvr 建立优质文章的索引，成功将优质文章的召回率在旧方案的基础上提高了 7%。此外，搜一搜利用 SimSvr 索引视频指纹向量进行相似视频去重，构建的单个 table 包含 1.7 亿 128 维的数据，平均召回延时不到 8ms，日检索量高达 12.5 亿。

此外，有本论文作者单独负责的 Balanced IVF-HNSW 部分截至 2021 年 3 月已经上线服务，索引的数据量达到了 20 亿。Balanced IVF-HNSW 的稳定性得到了生产数据的有力支撑。

5.4 本章小结

本章介绍了工业级的适用于超大规模数据的分布式近似最近邻搜索组件 SimSvr。它以 HNSW、IVF-HNSW 以及 Balanced IVF-HNSW 等算法为核心检索算法，具有性能好、召回率高、可扩展性高以及稳定性强等特性。SimSvr 已经广泛应用微信的看一看、搜一搜等核心业务，索引数据量达到数十亿的级别，证明 SimSvr 是一个优秀的大规模数据的近似最近邻检索的组件。

第六章　总结与展望

本文从算法应用的角度，探讨了在实践中应用近似最近邻算法需要关注的问题。在实际应用中，算法之间并不存在高下之分，适合于解决实际问题的算法就是好算法。针对特定应用场景，需要考虑两个方面的因素，一个是对算法本身的要求，另一个是要考虑当前计算资源的情况。对于算法本身，需要关注算法能够达到的召回率和召回时间是否满足需求。从计算资源的角度需要考虑当前内存、CPU 等资源是否足够，能够容纳多少数据等等。对这些问题的思考贯穿解决实际问题的始终。

从算法角度来说，本文有以下贡献：

(1) 提出了算法 HNSW Mutual-Remove。该算法用于高效地从 HNSW 构建的索引结构中删除节点，解决了 HNSW 原有节点删除算法可能导致部分搜索请求返回结果数量不足的问题。

(2) 提出了算法 Balanced IVF-HNSW。该算法为 IVF-HNSW 提出了新的索引构建算法，能够大幅度加快 IVF-HNSW 的构建速度。IVF-HNSW 能够大幅压缩构建的索引的大小，但是其构建索引过程比较漫长，不利于 IVF-HNSW 的大规模应用，Balanced IVF-HNSW 的提出让 IVF-HNSW 变得更加实用。

上述对算法的优化被成功应用到了微信大规模分布式近似最近邻搜索组件 SimSvr 中。其中 HNSW Mutual-Remove 提高了 SimSvr 整体的稳定性，这是由于该算法保证了每次搜索请求都能够返回期望数量的结果。Balanced IVF-HNSW 加快了索引的构建速度，有利于 SimSvr 对数据定期更新的支持。当然搜索算法只是 SimSvr 的一部分，SimSvr 目前的架构并不适合数据的大规模实时更新，这个可能是未来需要改进的方向。

展望未来，人工智能等技术仍然被广泛需求与应用，对相似特征检索的需求也在持续增加中。在算法层面，由于度量数据之间相似度的距离计算方式多种多样，但是并不是每种算法都对各种距离有很好的支持，探索更加通用的算法是一个研究方向。此外，IVF-HNSW 虽然能够大幅压缩构建的索引的大小，但是它需要复杂的调参，这限制了它的实用性。是否存在能够达到同样压缩效果

同时又简单易用的方法呢。在工程应用领域，目前并不存在一个通用的，可以直接分布式部署的支持数据实时更新的组件。当然已经有些组件正在达到接近的效果，这也是未来需要努力的方向。

参考文献

[1] GOODFELLOW I, BENGIO Y, COURVILLE A, et al. Deep learning: volume 1 [M]. [S.l.]: MIT press Cambridge, 2016.

[2] CLARKSON K L. An algorithm for approximate closest-point queries[C]// Proceedings of the tenth annual symposium on Computational geometry. [S.l.: s.n.], 1994: 160-164.

[3] BENTLEY J L. Multidimensional binary search trees used for associative searching[J]. Communications of the ACM, 1975, 18(9):509-517.

[4] Wikipedia contributors. Inverted index — Wikipedia, the free encyclopedia [EB/OL]. 2020. https://en.wikipedia.org/w/index.php?title=Inverted_index&oldid=992702992.

[5] Wikipedia contributors. Vector quantization — Wikipedia, the free encyclopedia [EB/OL]. 2020. https://en.wikipedia.org/w/index.php?title=Vector_quantization&oldid=993993570.

[6] Wikipedia contributors. Simd — Wikipedia, the free encyclopedia[EB/OL]. 2021. https://en.wikipedia.org/w/index.php?title=SIMD&oldid=1007613409.

[7] Owens J D, Houston M, Luebke D, et al. Gpu computing[J/OL]. Proceedings of the IEEE, 2008, 96(5):879-899. DOI: 10.1109/JPROC.2008.917757.

[8] INDYK P, MOTWANI R. Approximate nearest neighbors: towards removing the curse of dimensionality[C]//Proceedings of the thirtieth annual ACM symposium on Theory of computing. [S.l.: s.n.], 1998: 604-613.

[9] SUN Y, WANG W, QIN J, et al. Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index[J]. Proceedings of the VLDB Endowment, 2014.

[10] HUANG Q, FENG J, ZHANG Y, et al. Query-aware locality-sensitive hashing for approximate nearest neighbor search[J]. Proceedings of the VLDB Endowment, 2015, 9(1):1-12.

[11] SILPA-ANAN C, HARTLEY R. Optimised kd-trees for fast image descriptor matching[C]//2008 IEEE Conference on Computer Vision and Pattern Recognition. [S.l.]: IEEE, 2008: 1-8.

[12] FUKUNAGA K, NARENDRA P M. A branch and bound algorithm for computing k-nearest neighbors[J]. IEEE transactions on computers, 1975, 100(7):750-753.

[13] MALKOV Y A, YASHUNIN D A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs[J]. IEEE transactions on pattern analysis and machine intelligence, 2018.

[14] PUGH W. Skip lists: a probabilistic alternative to balanced trees[J]. Communications of the ACM, 1990, 33(6):668-676.

[15] ARYA S, MOUNT D M. Approximate nearest neighbor queries in fixed dimensions.[C]//SODA: volume 93. [S.l.]: Citeseer, 1993: 271-280.

[16] FU C, XIANG C, WANG C, et al. Fast approximate nearest neighbor search with the navigating spreading-out graph[J]. arXiv preprint arXiv:1707.00143, 2017.

[17] SUBRAMANYA S J, DEVVRIT F, SIMHADRI H V, et al. Rand-nsg: Fast accurate billion-point nearest neighbor search on a single node[C]//Advances in Neural Information Processing Systems. [S.l.: s.n.], 2019: 13771-13781.

[18] BARANCHUK D, BABENKO A, MALKOV Y. Revisiting the inverted indices for billion-scale approximate nearest neighbors[C]//Proceedings of the European Conference on Computer Vision (ECCV). [S.l.: s.n.], 2018.

[19] EDELSBRUNNER H. Algorithms in combinatorial geometry: volume 10[M]. [S.l.]: Springer Science & Business Media, 2012.

[20] MINSKY M, PAPERT S A. Perceptrons: An introduction to computational geometry[M]. [S.l.]: MIT press, 2017.

[21] JEGOU H, DOUZE M, SCHMID C. Product quantization for nearest neighbor search[J]. IEEE transactions on pattern analysis and machine intelligence, 2010, 33(1):117-128.

[22] Wikipedia contributors. Locality-sensitive hashing — Wikipedia, the free encyclopedia[EB/OL]. 2020. https://en.wikipedia.org/w/index.php?title=Locality-sensitive_hashing&oldid=985085303.

[23] DATAR M. Locality-sensitive hashing scheme based on p-stable distributions [C]//Proc. of the 20th ACM Symposium on Computational Geometry, 2004. [S.l.: s.n.], 2004.

[24] ZHAI J, LOU Y, GEHRKE J. Atlas: a probabilistic algorithm for high dimensional similarity search[C]//Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011. [S.l.: s.n.], 2011.

[25] WEISS Y, TORRALBA A, FERGUS R. Spectral hashing[J]. Advances in neural information processing systems, 2008, 21:1753-1760.

[26] LIU W, HE J, CHANG S F. Large graph construction for scalable semi-supervised learning[C]//International Conference on Machine Learning. [S.l.: s.n.], 2010.

[27] JIANG Q Y, LI W J. Scalable graph hashing with feature transformation.[C]// IJCAI: volume 15. [S.l.: s.n.], 2015: 2248-2254.

[28] NAIDAN B, BOYTSOV L, NYBERG E. Permutation search methods are efficient, yet faster search is possible[J]. arXiv preprint arXiv:1506.03163, 2015.

[29] LLOYD S. Least squares quantization in pcm[J]. IEEE transactions on information theory, 1982, 28(2):129-137.

[30] FORGY E W. Cluster analysis of multivariate data: efficiency versus interpretability of classifications[J]. biometrics, 1965, 21:768-769.

[31] Wikipedia contributors. Delaunay triangulation — Wikipedia, the free encyclopedia[EB/OL]. 2020. https://en.wikipedia.org/w/index.php?title=Delaunay_triangulation&oldid=987950231.

[32] Wikipedia contributors. Nearest neighbor graph — Wikipedia, the free encyclopedia[EB/OL]. 2020. https://en.wikipedia.org/w/index.php?title=Nearest_neighbor_graph&oldid=956578789.

[33] Wikipedia contributors. Small-world network — Wikipedia, the free encyclopedia [EB/OL]. 2020. https://en.wikipedia.org/w/index.php?title=Small-world_network&oldid=991831407.

[34] AUMÜLLER M, BERNHARDSSON E, FAITHFULL A. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms[J]. Information Systems, 2020, 87:101374.

[35] MALKOV Y, PONOMARENKO A, LOGVINOV A, et al. Approximate nearest neighbor algorithm based on navigable small world graphs[J]. Information Systems, 2014, 45:61-68.

[36] PUGH W. Skip lists: a probabilistic alternative to balanced trees[J]. Communications of the ACM, 1990, 33(6):668-676.

[37] JOHNSON J, DOUZE M, JÉGOU H. Billion-scale similarity search with gpus [J]. arXiv preprint arXiv:1702.08734, 2017.

[38] GE T, HE K, KE Q, et al. Optimized product quantization for approximate nearest neighbor search[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. [S.l.: s.n.], 2013: 2946-2953.

[39] Boufounos P T. Universal rate-efficient scalar quantization[J/OL]. IEEE Transactions on Information Theory, 2012, 58(3):1861-1872. DOI: 10.1109/TIT.2011.2173899.

[40] LOWE D G. Distinctive image features from scale-invariant keypoints[J]. International Journal of Computer Vision, 2004, 60(2):91-110.

[41] FRIEDMAN, ALINDA. Framing pictures: The role of knowledge in automatized encoding and memory for gist.[J]. Journal of Experimental Psychology: General, 1979, 108(3):316-355.

[42] Wikipedia contributors. Advanced vector extensions — Wikipedia, the free encyclopedia[EB/OL]. 2021. https://en.wikipedia.org/w/index.php?title=Advanced_Vector_Extensions&oldid=998929503.

[43] Wikipedia contributors. Memory hierarchy — Wikipedia, the free encyclopedia [EB/OL]. 2021. https://en.wikipedia.org/w/index.php?title=Memory_hierarchy&oldid=1002366093.

[44] Wikipedia contributors. Cpu cache — Wikipedia, the free encyclopedia[EB/OL]. 2021. https://en.wikipedia.org/w/index.php?title=CPU_cache&oldid=1008782388.

[45] JÉGOU H, TAVENARD R, DOUZE M, et al. Searching in one billion vectors: re-rank with source coding[C]//2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). [S.l.]: IEEE, 2011: 861-864.

[46] SCULLEY D. Web-scale k-means clustering[C]//Proceedings of the 19th international conference on World wide web. [S.l.: s.n.], 2010: 1177-1178.

简历与科研成果

基本信息

刘凤山，男，汉族，1995 年 9 月出生，湖北省恩施市人。

教育背景

2018 年 9 月 — 2021 年 6 月 南京大学计算机科学与技术系 硕士

2014 年 9 月 — 2018 年 6 月 电子科技大学信息与通信工程学院 本科

攻读硕士学位期间完成的学术成果

1. An, Junyi, **Fengshan Liu**, Jian Zhao, and Furao Shen "IC Neuron: An Efficient Unit to Construct Neural Networks", in arXiv preprint arXiv:2011.11271 (2020).
2. An, Junyi, **Fengshan Liu**, Jian Zhao, and Furao Shen "C Networks: Remodeling the Basic Unit for Convolutional Neural Networks", in arXiv preprint arXiv:2102.03495 (2021).

攻读硕士学位期间的发明专利

1. 申富饶, **刘凤山**, 赵健, 李俊. ''一种增量式语音命令词识别方法"(201911080670.8)
2. 申富饶, **刘凤山**. ''一种基于激光雷达的液压支架对齐方法"(202010326088.1)

攻读硕士学位期间参与的科研课题

1. 国家自然科学基金面上项目 ''基于深度感知增量式联想记忆神经网络的信息融合系统研究，Information fusion system based on deep perception and incremental associative memory neural networks"（课题年限 2019.01 ~ 2022.12），负责增量学习算法研究。

其他成果

1. github 开源项目 hnswlib 主要贡献者之一。

致　　谢

三年的研究生生活即将结束。在此我想感谢一路上帮助过我的人。

首先要感谢我的导师申富饶老师。在学术上，申老师是一个治学严谨，知识渊博的人。申老师在课题选择、分析问题与解决问题方面给予了悉心的教导。他会让我思考为什么要做研究，为了解决什么问题而做研究，很多时候让我避免走进死胡同。生活中，申老师是一个十分和善的人，会给我传授他的人生经验，让我少走很多弯路。

其次感谢一起进入 RINC 研究组的 18 级同学以及我的室友对我在学习和生活上的帮助。感谢 sauron 与 flash，在他们的带领下我参与了 SimSvr 这个项目，收获良多。

最后感谢我的父母，感谢他们多年来对我无私的支持。

学位论文出版授权书

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》（以下简称“章程”），愿意将本人的学位论文提交“中国学术期刊（光盘版）电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名：_____

_____年_____月_____日

论文题名	近似最近邻搜索算法研究与应用				
研究生学号	MG1833098	所在院系	计算机科学与技术系	学位年度	2021
论文级别	☑ 硕士　☐ 硕士专业学位 ☐ 博士　☐ 博士专业学位　（请在方框内画勾）				
作者 Email	liufengshan@smail.nju.edu.cn				
导师姓名	申富饶 教授				

论文涉密情况：

☑ 不保密

☐ 保密，保密期：_____年_____月_____日 至 _____年_____月_____日

注：请将该授权书填写后装订在学位论文最后一页（南大封面）。