第 18 章 向量检索系统

在前面的章节中，我们重点介绍了推荐系统工程实践中的离线服务与在线服务。离线服务负责完成样本构建、特征处理以及模型训练等任务，使海量数据能够在分布式环境下高效流转；在线服务则承担模型推理与结果返回，通过高性能计算框架保证推荐系统在毫秒级延迟下稳定响应。正是离线训练与在线服务的协同配合，构成了工业级推荐系统的基础设施。

然而，仅有模型训练与推理还不足以支撑大规模推荐系统。在实际业务中，无论是级联式推荐架构中的召回阶段，还是用户行为序列建模中的相似兴趣查找，本质上都需要解决一个核心问题：

笔记 给定一个用户向量，在数亿甚至数十亿个候选向量中，如何快速找到与其最相似的 Top-K 个向量？

这一问题被称为**向量检索（Vector Retrieval）**问题，也是现代推荐系统、搜索系统以及大模型 RAG 系统中的核心基础能力。而向量检索系统正是用于解决这一问题。从本质上来看，向量检索系统是一种高性能相似度计算服务。它能够在极低延迟下完成海量向量之间的相似度搜索，从而实现内容召回、用户兴趣匹配以及知识检索等功能。因此，向量检索系统实际上也是推荐系统在线服务体系的重要组成部分，其目标是在有限计算资源下，同时满足**高吞吐、低延迟以及高召回率**等要求。

本章将系统介绍工业界广泛使用的向量检索技术，重点讨论近似最近邻搜索（Approximate Nearest Neighbor Search，ANN）的基本思想，并进一步介绍基于树结构、基于哈希算法、基于向量量化以及基于图的经典 ANN 算法等，以及 Faiss、ScaNN 这些工业级向量检索系统的实现原理。

18.1 ANN 概述

在工业级推荐系统中，向量召回 (Embedding-based Retrieval) 已经成为当前主流的召回范式。向量召回的基本思想如下：

定义 18.1

向量召回本质上是利用深度学习模型将用户和物品映射到统一的低维稠密向量空间中，并通过向量之间的距离或相似度来衡量用户对物品的兴趣程度。

这里我们举一个简单的例子。比如一个用户观看过大量篮球比赛视频，那么经过双塔模型编码后，可以得到一个用户向量：

$$\mathbf{u} = [0.21, -0.35, 0.84, \cdots] \tag{18.1}$$

同时，每个视频也会对应一个物品向量：

$$\mathbf{v}_i = [0.19, -0.31, 0.80, \cdots] \tag{18.2}$$

在线推荐时，只需要找到与用户向量最接近的若干个视频向量，即可完成候选集的召回。从直观上来看，这一过程类似于在一座拥有数十亿本书的图书馆中寻找与当前书籍内容最相似的几本书。一种最直观的想法就是将当前书籍与图书馆中的每一本书逐一进行比较，计算相似度后再排序选出 Top-K。这种方法被称为**精确最近邻搜索（Exact Nearest Neighbor Search）**。

但是，在工业级推荐系统中，候选物品全库的数量可能在数亿甚至数十亿的量级，精确最近邻搜索的速度一定是非常慢的。这里我们假设物品库规模为 $N = 10^9$，向量维度为 $d = 128$，那么一次查询需要进行约 $N \times d \approx 10^{11}$ 次浮点运算，即使在高性能服务器上，也难以满足在线服务毫秒级响应的要求。因此，我们会很自然地想到如下所示的问题：

笔记 是否能够不遍历所有向量，而是通过某种方法快速定位到可能包含最近邻的区域，仅访问极少部分候选向量，从而获得近似最优结果？

这便引出了**近似最近邻搜索（Approximate Nearest Neighbor Search，ANN）**。ANN 与精确最近邻搜索的区

别如下图18.1所示：

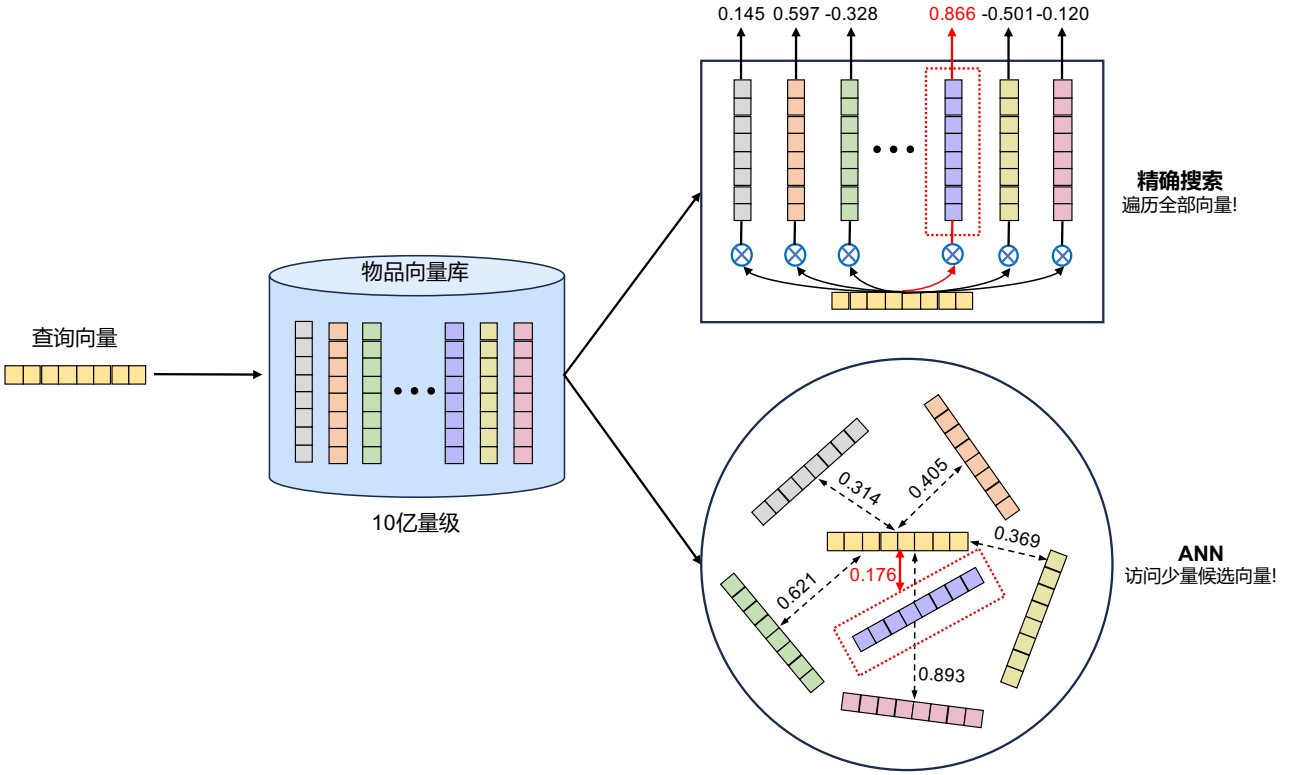

图 18.1: ANN 与精确搜索的对比。

从上面的示意图可以简要地看出，ANN 是对精确搜索的一种加速化处理。ANN 的核心思想如下：

定义 18.2

ANN 希望在牺牲极少量的搜索精度的情况下，以数十倍甚至数百倍的速度提升整个向量检索速度。因此，ANN 需要通过各种数据结构与算法的优化来降低整个算法的时间复杂度。

例如，在十亿级向量库中，精确最近邻搜索方法可能需要访问全部 10^9 个向量，而 ANN 方法只需要访问其中几万个甚至几千个候选，就可以获得 95% 以上的召回率，从而将查询延迟从秒级降低到毫秒级。需要强调的是，ANN 并不是一种具体算法，而是一类高效近似搜索方法的统称，不同的 ANN 算法会采用不同的策略来缩小搜索的范围。

目前 ANN 的主流方法大致可以分为以下几类：

- **树结构方法（Tree-based）**：利用 KD 树、随机投影树（Random Projection Tree）等结构递归划分空间，通过树搜索快速定位候选区域，代表系统包括 Annoy。
- **哈希方法（Hash-based）**：通过局部敏感哈希（Locality Sensitive Hashing，LSH）将相似向量映射到相同桶中，从而减少比较次数。
- **量化方法（Quantization-based）**：通过向量压缩降低存储与计算成本，典型代表包括 IVF、PQ、OPQ 等方法，也是 Faiss 中应用最广泛的一类技术。
- **图结构方法（Graph-based）**：构建向量之间的邻接关系图，通过贪心搜索逐步逼近目标向量，代表算法包括 HNSW、NSG 等，目前已成为工业界最主流的方案之一。

近年来，多个高性能 ANN 系统相继出现，并将上述不同类型的 ANN 算法进行了工程化的封装。如下所示是目前工业界使用较多的几种 ANN 算法库。

- **Annoy（Approximate Nearest Neighbors Oh Yeah）**：由 Spotify 开源，基于随机投影树，具有轻量、易部署的特点，适用于中小规模场景。

- **Faiss**：由 Meta 开发，支持 IVF、PQ、OPQ、HNSW 等多种索引结构，并提供 GPU 加速能力，是目前工业界应用最广泛的向量检索框架之一。
- **ScaNN（Scalable Nearest Neighbors）**：由 Google Research 提出，通过各向异性向量量化（Anisotropic Vector Quantization）和高度优化的 CPU 指令实现极高的检索效率，特别适用于超大规模在线服务场景。

这些系统之所以能够在亿级甚至十亿级向量库中实现毫秒级检索，其核心依赖于底层高效的 ANN 算法。接下来，我们将根据上述 ANN 算法的分类分别对底层 ANN 算法原理进行详细的介绍，最后我们会对这些常见 ANN 算法库的使用进行说明并给出相应的使用示例。

18.2 基于树结构的 ANN

树结构方法（Tree-based ANN）是最早被应用于近似最近邻搜索的一类方法，其核心思想是利用树结构递归地划分向量空间，从而避免对全部向量进行暴力遍历。假设向量库中共有 N 个样本，每个样本维度为 d。若采用精确最近邻搜索（Exact Nearest Neighbor Search），则需要计算查询向量与所有样本之间的距离，其时间复杂度为 $\mathcal{O}(ND)$。当 N 达到百万甚至亿级时，这种暴力搜索难以满足在线系统毫秒级响应的要求。而树结构方法通过预先建立空间索引，在查询阶段只访问部分区域，从而将搜索复杂度降低至近似 $\mathcal{O}(\log N)$。当然，由于只搜索部分节点，因此检索结果不一定是真正的最近邻，而是在召回率（Recall）和查询速度（Latency）之间进行权衡。

树结构方法主要包括 KD 树 (KD Tree)、球树 (Ball Tree) 以及随机投影树（Random Projection Tree）等。下面我们分别对这几类方面进行详细的介绍说明。

18.2.1 KD 树

KD 树（K-Dimensional Tree, KD Tree）最早由 Bentley 在 1975 年发表的论文《Multidimensional Binary Search Trees Used for Associative Searching》[1] 中提出，它是一种针对 K 维空间设计的二叉搜索树。

KD 树的核心思想如下：

定义 18.3

KD 树在构建过程中不断选择某一个维度作为划分轴，并利用当前维度上的中位数将整个空间递归地划分为两个子空间，从而形成一棵二叉树结构。最终，每个叶节点对应空间中的一个超矩形（Hyper-Rectangle）。♣

18.2.1.0.1 KD 树建树 下面以 KD 树原始论文中的二维示例说明其构建过程。如图 18.2 所示，假设二维空间中存在如下七个样本点：

$$A:(50,50),\quad B:(10,70),\quad C:(80,15),\quad D:(25,20),\quad E:(40,85),\quad F:(70,85),\quad G:(10,60) \tag{18.3}$$

KD 树的构建过程实际上就是不断利用不同坐标轴对空间进行递归划分。首先，在根节点处按照 x 坐标进行划分。将所有点按照横坐标排序：

$$10,\ 10,\ 25,\ 40,\ 50,\ 70,\ 80 \tag{18.4}$$

其中中位数对应点 $E(40,85)$。这里为了和原始 KD 树论文示例保持一致，我们选择点 A 作为根节点，并生成一条穿过点 A 的竖直分割线，将整个二维空间划分为左右两个区域：

- 左侧区域：$\{B,D,E,G\}$；
- 右侧区域：$\{C,F\}$。

随后，对左右两个子区域继续递归划分，但此时划分维度切换为 y 坐标。对于左侧区域中的四个点：

$$20,\ 60,\ 70,\ 85 \tag{18.5}$$

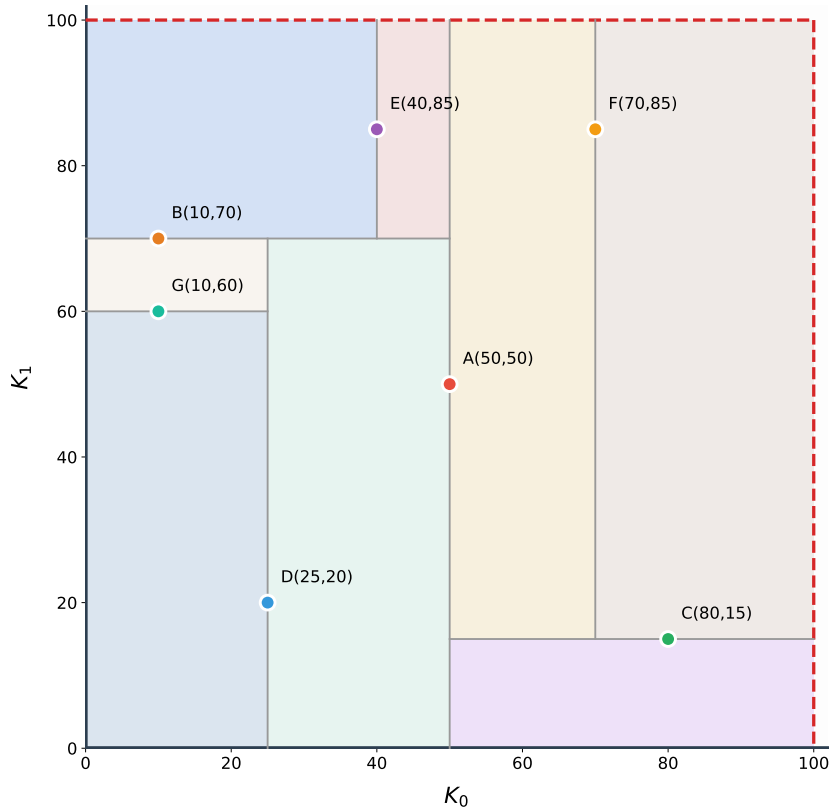

图 18.2: 二维空间中的 KD 树划分过程。

按照纵坐标排序后，可以选择点 $B(10,70)$ 作为左子树根节点，并产生一条经过点 B 的水平分割线，将该区域进一步划分。同理，在右侧区域中，根据 y 坐标可以选择点 $C(80,15)$ 作为右子树根节点。

接下来再切换回 x 坐标，对更小的子区域继续划分。如此不断在不同坐标轴之间循环切换，直到叶节点中只包含一个样本点或者达到停止条件。经过递归划分之后，整个二维平面被分割成若干互不重叠的矩形区域，而这些空间划分过程实际上对应于一棵二叉树结构。需要注意的是，KD 树构建过程通常遵循如下两个原则：

1. 在当前划分维度上选择中位数对应的数据点作为划分点，从而尽可能保证左右子树平衡；
2. 划分维度按照坐标轴轮流切换，使得任意连续的 K 层划分中，每一个维度都能够被使用一次，从而避免树结构过度偏斜。

上述二维空间划分过程对应的 KD 树结构如图 18.3 所示。

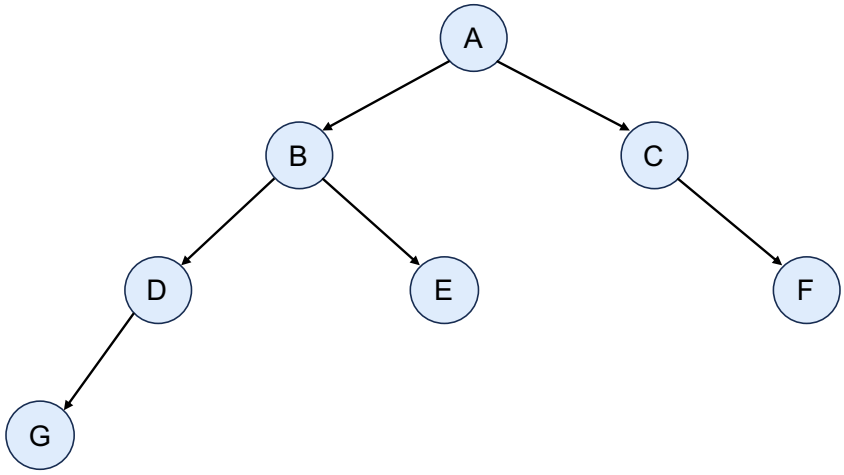

图 18.3: 二维样本对应的 KD 树结构。

可以看到，每一个内部节点对应一个划分点，同时记录当前的划分维度；左子树和右子树分别对应超平面两侧的数据点集合。因此，空间划分过程与二叉树结构是一一对应的。由于每次建树的时候都尽量以当前坐标维度下数据点的中位数作为分割点，因此树的高度大约满足 $\mathcal{O}(\log n)$，而每一层建树的时候寻找中位数，实际上在 $\mathcal{O}(n)$ 的时间复杂度内就能完整，而不需要直接对当前层的数据进行全局 sort 排序之后再找中位数，这样每一层的复杂度会变成 $\mathcal{O}(n \log n)$。最终，KD 树建树的时间复杂度最快可达到 $\mathcal{O}(\log n) \cdot \mathcal{O}(n) = \mathcal{O}(n \log n)$ 的水平。

18.2.1.0.2 KD 树查询 在 KD 树建树完成之后，就可以基于已有的 KD 树进行高效的查询操作。KD 树的查询采用“向下搜索 + 回溯剪枝”的策略。假设查询向量为 q，首先从根节点开始，根据当前划分维度判断查询点位于分割超平面的哪一侧，然后沿着对应分支不断向下搜索，直到到达某个叶节点。

此时，我们可以得到当前找到的最近邻点，其距离记为

$$r = d(q, x_{\text{nearest}}) \tag{18.6}$$

随后开始向上回溯。对于每一个父节点，需要判断其兄弟子树对应的超矩形区域是否可能包含更近的样本。设当前节点的划分超平面为

$$x_i = s \tag{18.7}$$

其中 i 表示划分维度，s 为划分值。若满足

$$|q_i - s| > r \tag{18.8}$$

则说明查询点到超平面的距离已经大于当前最近邻距离，因此另一侧区域中的所有样本都不可能比当前最近邻更近，可以直接剪枝。反之，若满足

$$|q_i - s| < r \tag{18.9}$$

则兄弟节点对应区域中仍然可能存在更近的样本，需要继续访问。通过这种回溯剪枝机制，KD 树避免了对全部样本进行遍历。若树结构较为平衡，则平均查询复杂度约为 $\mathcal{O}(\log N)$。因此，在二维、三维等低维空间中，KD 树能够获得非常高的搜索效率。

18.2.1.0.3 维度灾难 虽然 KD 树在低维空间中能够通过空间划分实现高效剪枝，但随着向量维度不断增加，其性能会迅速下降。例如，在推荐系统中，Embedding 向量通常具有 64、128、256 甚至更高维度。高维空间中 KD 树性能下降的根本原因在于，随着维度增加，样本点会变得越来越稀疏，使得查询点到最近邻的距离逐渐增大。

在回溯过程中，KD 树需要利用当前最近邻距离 r 对兄弟节点对应的超矩形区域进行剪枝。设超矩形区域为 R，若

$$dist(q, R) > r \tag{18.10}$$

则整个区域可以被剪枝掉，不再参与后续的回溯搜索过程。然而在高维空间中，由于最近邻距离不断增大，大量超矩形区域都会满足

$$dist(q, R) \leq r \tag{18.11}$$

从而导致越来越多的节点无法被剪枝。此时，需要访问的节点数会随着维度增加迅速增长，在极端情况下甚至趋近于全部节点，使得搜索复杂度重新退化为 $\mathcal{O}(N)$。换句话说，KD 树在这种情况下会逐渐失去相对于暴力搜索的加速优势，而这种现象被称为**维度灾难（Curse of Dimensionality）**。

总体而言，KD 树更适用于低维（通常 $D < 20$）、数据规模较小的场景。而对于推荐系统中常见的高维大规模 Embedding 检索任务，其性能会急剧下降，因此现代工业级向量检索系统已经很少直接使用 KD 树了，而更多采用随机投影树、乘积量化（PQ）以及图结构（HNSW）等更加高效的方法。

18.2.2 球树

为了缓解 KD 树对坐标轴划分过于依赖的问题，球树 (Ball Tree) 使用超球体（Hyper-Sphere）而非超矩形来划分空间。球树最早由 Stephen M. Omohundro 在 1989 年发表的技术报告《Five Balltree Construction Algorithms》[8] 中提出。

球树的核心思想如下：

定义 18.4

在球树中，每个节点对应一个超球体：

$$B(c, r) = \{x : \|x - c\| \le r\} \tag{18.12}$$

其中 c 表示球心（pivot），r 表示半径（radius）。父节点对应的超球体覆盖其所有子节点的超球体，而叶节点中存储实际的数据点集合。

♣

18.2.2.0.1 球树的构建 Omohundro 在原始论文中提出了五种球树构建方式，其中基于递归划分（recursive partition）的策略由于时间复杂度较低（约 $\mathcal{O}(n \log n)$）而在工程中应用最为广泛，例如 scikit-learn 等库也采用类似策略。球树的构建过程与 KD 树类似，也需要将当前节点的数据集划分为两个子集，但划分依据不再是坐标轴切分，而是基于几何结构或统计投影进行分裂。常见方法包括：

- 选择方差最大的方向进行投影，并按中位数划分；
- 基于质心（centroid）或最大最小值中心进行启发式划分；
- 使用 k-means 或类似聚类方法进行二分划分。

随后递归构建左右子球，直到满足停止条件（如叶子节点样本数小于 leaf size）。

下面我们继续沿用 KD 树中的二维示例数据进行说明。假设数据点为：

$$A : (50, 50), \quad B : (10, 70), \quad C : (80, 15), \quad D : (25, 20), \quad E : (40, 85), \quad F : (70, 85), \quad G : (10, 60) \tag{18.13}$$

首先在所有样本上选择方差较大的维度（例如 x 轴）进行投影排序：

$$10,\ 10,\ 25,\ 40,\ 50,\ 70,\ 80 \tag{18.14}$$

取中位数附近作为划分基准点，将数据划分为两组：

- 左子集：$\{B, D, E, G\}$
- 右子集：$\{A, C, F\}$

随后对左子集与右子集分别递归进行划分。例如对左子集在 y 轴上再次投影排序：

$$20,\ 60,\ 70,\ 85 \tag{18.15}$$

进一步划分为 $\{B, E\}$ 和 $\{D, G\}$ 两个子集。右子集同理递归划分，直到每个叶子节点满足停止条件。最终得到的球树划分结果如图 18.4 所示。

需要强调的是，与 KD 树通过超平面显式划分空间不同，球树并不存在严格意义上的“左右空间划分”。其构建过程本质上是对当前节点的数据点集合进行划分（partition），划分结果形成两个子集合，并分别递归构造子球。因此，“左子树”和“右子树”仅表示递归结构中的两个子数据集合，而不对应空间中的几何方向。

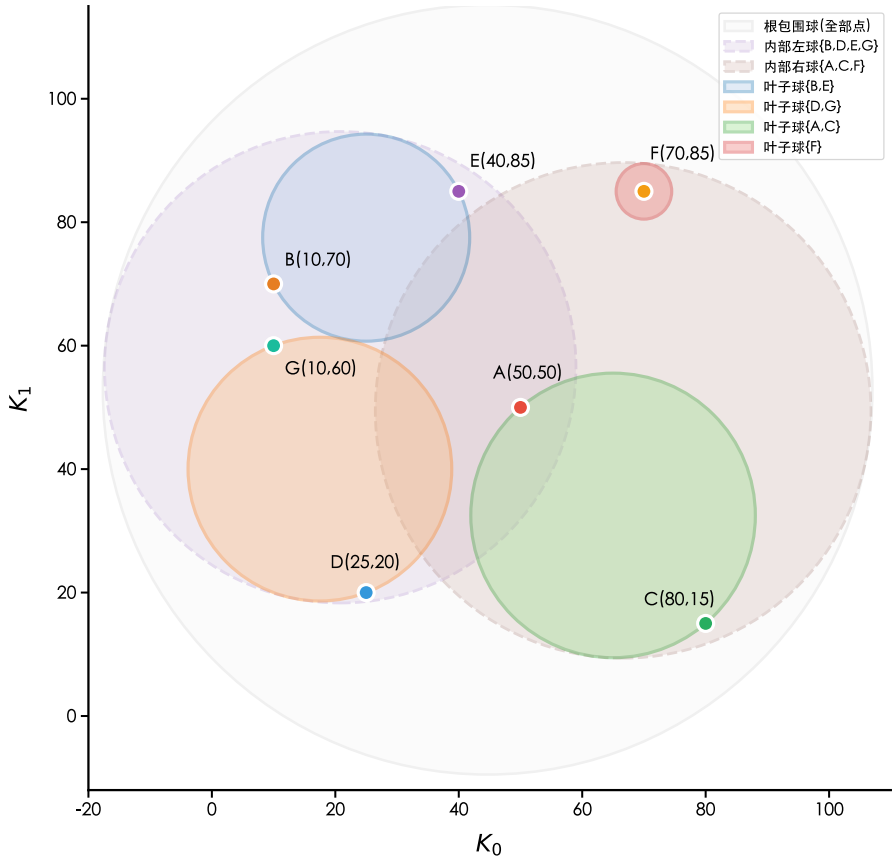

图 18.4: 二维空间中球树划分示例。

18.2.2.0.2 叶子节点与内部节点 为了平衡球树的搜索遍历速度以及球树的内存存储占用情况，在实际工程中，球树通常不会将叶子节点划分到单个样本，而是设定叶子大小 leaf_size（例如 20 或 40）。当当前子集大小满足：

$$n \leq \text{leaf\_size} \tag{18.16}$$

则停止继续划分，并直接构造叶子节点所对应的超球体。

假设球树叶子节点对应的数据样本点为：

$$\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\} \tag{18.17}$$

则叶子节点的球心通常取样本均值：

$$\boldsymbol{c} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x}_i \tag{18.18}$$

半径定义为：

$$r = \max_{i=1,\ldots,n} \|\boldsymbol{x}_i - \boldsymbol{c}\|_2 \tag{18.19}$$

对于内部节点，其球体由左右子球合并得到。设左、右子球参数分别为：

$$(c_L, r_L), \quad (c_R, r_R) \tag{18.20}$$

则首先计算两个球心之间的距离：

$$d = \|c_L - c_R\|_2 \tag{18.21}$$

如果满足一个球完全包含另一个球，即：

$$d + \min(r_L, r_R) \leq \max(r_L, r_R) \tag{18.22}$$

则父节点可以直接复用较大的子球：

$$\boldsymbol{c} = \boldsymbol{c}_{\text{big}}, \quad r = r_{\text{big}} \tag{18.23}$$

否则，就会根据两个子球来构造最小包围球，此时父球的球心（两球心连线上偏移中点）计算如下：

$$\boldsymbol{c}_{\text{parent}} = \boldsymbol{c}_L + \frac{d + r_L - r_R}{2d}(\boldsymbol{c}_R - \boldsymbol{c}_L) \tag{18.24}$$

而父球半径（两球最外侧端点间距的一半）为：

$$r_{\text{parent}} = \frac{d + r_L + r_R}{2} \tag{18.25}$$

总的来说，球树中叶子节点球心是由真实样本统计均值生成，和数据集强相关。而内部节点球心是纯几何运算得到的虚拟中心点，不能保证对应任意真实数据样本。

18.2.2.0.3 球树与 KD 树的对比 这里，我们需要解答一下以下的疑问，即：

笔记 为什么要用球树来替代 KD 树呢？本质原因是什么？

下面我们通过一个例子看 KD 树与球树在 2D 空间中的距离计算的对比，如下图18.5所示。图中的查询点坐标是 $(7,3)$，而 KD 树中的四个坐标点 $(1,3)$、$(2.5,6)$、$(3,2)$ 和 $(4,4)$ 构成的矩形范围是 $[1,4]\times[2,6]$，球树构成了以 $(2.5,4.0)$ 为圆心，半径为 2.5 的圆形。从图中可以看出，查询点与 KD 树节点对应矩形区域的距离计算通

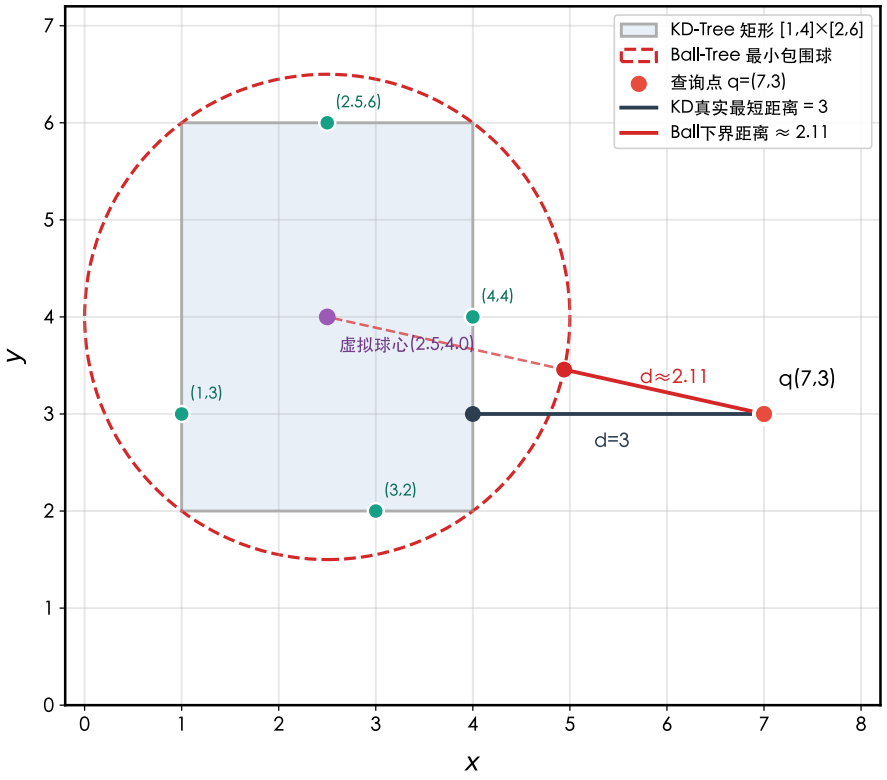

图 18.5: 2D 空间 KD 树与球树距离计算对比。

常会计算当前查询坐标到矩形区域边缘的欧式距离，即 $d_{\text{kd}} = 3.0$。而查询点与球树节点对应的圆形区域的距离计算实际上是计算查询点到圆形边缘的距离，即 $d_{\text{ball}} \approx 2.11$。

我们将数据点的维度向高维扩展。假设查询点为 $\mathbf{q} = [q_1, \cdots, q_D]$，KD 树中某个节点对应的超矩形为：

$$\mathbf{R} = [l_1, u_1] \times \cdots \times [l_D, u_D] \tag{18.26}$$

这里 $\mathbf{R}$ 表示超矩形，D 表示超矩形的最高维度。那么对于超矩形的每一维，我们就可以计算查询点的每一维与

超矩形的距离：

$$\delta_i = \begin{cases} l_i - q_i, & q_i < l_i \\ 0, & l_i \le q_i \le u_i \\ q_i - u_i, & q_i > u_i \end{cases} \tag{18.27}$$

最后就可以计算查询点与超矩形的距离：

$$d(\mathbf{q}, \mathbf{R}) = \sqrt{\sum_{i=0}^{D} \delta_i} \tag{18.28}$$

在查询点到 KD 树超矩形区域距离的计算过程中，我们发现超矩形每一个维度的上下界都需要参与计算，因此参与计算的边界点数据比较多，达到了 $2D$。而相比起来，计算查询点到球树超球体的距离是，只需要用到超球体的球心参与计算，即：

$$d(\mathbf{q}, \mathbf{B}) = d(\mathbf{q}, \mathbf{c}) - r \tag{18.29}$$

其中 $\mathbf{B}$ 表示球树节点对应的超球体，$\mathbf{c}$ 表示超球体的球心，而 r 表示超球体的半径。

从上述距离计算的对比来看，球树的距离计算相比起 KD 树还是要稍微简单一些，这源自于超球体有比较好的几何性质。除此之外，假设球树节点对应的超球体是 $\mathbf{B}(\mathbf{c}, r)$，我们可以利用三角不等式得到如下 KD 树不具备的性质，即：

$$\forall \mathbf{x} \in \mathbf{B}, \quad d(\mathbf{q}, \mathbf{x}) \ge d(\mathbf{q}, \mathbf{c}) - r \tag{18.30}$$

所以我们可以很容易得到查询点到超球体 $\mathbf{B}(\mathbf{c}, r)$ 的距离下界，即：

$$d_{\mathrm{LB}} = d(\mathbf{q}, \mathbf{c}) - r \tag{18.31}$$

这个距离下界的计算对于欧氏距离、余弦距离、以及其他任意满足三角不等式的度量，都可以非常自然地得出。

相比起球树使用的超球体这种几何性质更好的结构，KD 树中的超矩形在高维空间中会有非常多的角点，且每个超矩形也会变得非常细长。这样会导致的问题就是大量的高维超矩形会与搜索的区域相交，从而使得搜索剪枝的效率显著下降，比如有很多节点满足：

$$d(\mathbf{q}, \mathbf{R}) < d_{\mathrm{best}} \tag{18.32}$$

这里 d_{best} 是目前 kd 树搜索过程中记录到的最短距离。在这种情况下，kd 树查询就需要回溯太多的节点，导致搜索效率降低。因此，KD 树与球树相比，超矩形在高维空间中剪枝效率存在比较大的问题，这也体现了球树的优势所在。

这里我们从上述的示例和讨论中观察到 KD 树和球树的两个区别：

笔记 球树与 KD 树的第一个区别是：球树里面的圆心或者球心可能并不是真正的数据点，而可以是一个虚拟的点。KD 树需要基于数据点为分界线去按照坐标轴进行划分，所以 KD 树节点都是真实数据点。此外，球树与 KD 树的第二个区别是：KD 树的数据点都落在在超矩形分割超平面这种边界上，而球树的数据点都落在超球体的内部。

正是因为球树对于球心的定义不再局限于真实数据点，所以球心 c 可以是该节点样本集合的质心（centroid）、聚类中心或者其他人为定义的参考点。唯一需要满足的条件是节点中的所有样本点都位于以 c 为中心、半径为 r 的超球内部。这样看来，球树的灵活性也比 KD 树高了许多。与 KD 树利用真实样本点作为划分节点不同，球树更关注几何空间中的覆盖关系，而非样本点之间的拓扑结构。

18.2.2.0.4 球树查询 球树的查询过程基于“分治 + 剪枝”的思想，它的核心工具是三角不等式。对于任意节点对应球体 $\mathbf{B}(\mathbf{c}, r)$，以及查询点 $\mathbf{q}$，可以得到：

$$\|\mathbf{q} - \mathbf{x}\| \ge \|\mathbf{q} - \mathbf{c}\| - \|\mathbf{x} - \mathbf{c}\| \tag{18.33}$$

由于 $\|\mathbf{x}-\mathbf{c}\| \le r$，因此得到距离下界：

$$d_{\text{LB}}(\mathbf{q},\mathbf{B}) = \max(\|\mathbf{q}-\mathbf{c}\| - r, 0) \tag{18.34}$$

该下界表示的含义是查询点 q 到该球内任意点的最小可能距离。在实际查询过程中，球树使用如下所示的“优先搜索 + 剪枝回溯”策略：

- 从根节点开始递归向下搜索最近的叶子节点；
- 记录当前最优距离 d_{best}；
- 回溯过程中计算每个节点的下界 D_{LB}；
- 若 $D_{\text{LB}} > d_{\text{best}}$，则剪枝该子树；
- 否则继续访问。

为了提升剪枝效率，球树在实现中通常会引入“父节点下界继承”：

$$d^s(\mathbf{q}) = \begin{cases} \max\left(\|\mathbf{q}-\mathbf{c}_s\| - r_s, d^{s_{\text{parent}}}(\mathbf{q})\right), & s \neq \text{Root} \\ \max\left(\|\mathbf{q}-\mathbf{c}_s\| - r_s, 0\right), & s = \text{Root} \end{cases} \tag{18.35}$$

其中 $\mathbf{c}_s$ 表示球树节点 s 对应的超球体球心，r_s 表示球树节点 s 对应的超球体半径，$d^{s_{\text{parent}}}(\mathbf{q})$ 表示查询点 $\mathbf{q}$ 到球树节点 s 的父节点 s_{parent} 的距离。由于父节点 s_{parent} 通常对应半径更大的超球体，而球树节点 s 对应的超球体完全在父节点 s_{parent} 对应超球体的内部，所以距离 $d^s(\mathbf{q})$ 的下界是 $d^{s_{\text{parent}}}(\mathbf{q})$。

尽管球树在高维空间中相比起 KD 树具有更好的几何适应性，但随着维度继续增加，距离集中现象仍然会导致剪枝效率下降。因此在工业级推荐系统中，球树通常也不会作为主流方案，而是更多被 HNSW、IVF、PQ 等方法替代。

18.2.3 随机投影树

标准 KD 树始终沿坐标轴方向递归划分空间，在二维、三维等低维场景下能够获得较高的查询效率。然而，当数据维度不断增加时，KD 树会逐渐受到维度灾难（Curse of Dimensionality）的影响，其空间划分能力迅速下降。前文已经介绍，高维空间中的 Embedding 数据往往十分稀疏，KD 树对应的超矩形之间容易产生大量重叠区域，使得搜索过程中需要频繁回溯兄弟节点，最终查询复杂度逐渐退化为线性搜索。从更深层次来看，其根本原因在于 KD 树始终沿坐标轴进行划分，每次划分仅利用一个维度的信息，而高维数据通常分布在远低于环境维度（Ambient Dimension）的低维流形（Low-dimensional Manifold）上，固定坐标轴划分难以充分利用数据真实的几何结构。

例如，对于某些特殊构造的数据集，KD 树需要沿每一个坐标轴分别完成一次划分，才能使当前节点对应单元格（Cell）的直径缩小一半。若数据维度为 D（比如 $D=1000$），则理论上需要经历约 D 层划分才能达到这一目标；而为了构造这种最坏情况的数据集，其样本数量甚至需要达到 2^D（比如 $2^{1}000$）数量级。当维度较高时，无论是树结构存储还是查询计算都会产生巨大的开销，因此 KD 树并不适用于推荐系统中常见的高维 Embedding 检索任务。

针对这一问题，Dasgupta 和 Freund 在论文《Random Projection Trees and Low Dimensional Manifolds》[3] 中提出了随机投影树（Random Projection Tree，简称 RP-Tree）。与 KD 树不同，RP-Tree 不再沿固定坐标轴进行划分，而是在每个节点随机选择一个方向作为划分轴，使树结构能够更加自然地适应高维数据的几何分布，其理论收敛速度主要由数据的内在维数（Intrinsic Dimension）决定，而不再依赖于原始空间维数。

RP-Tree 的核心思想十分简单，可以概括为：

定义 18.5

随机投影树（Random Projection Tree）在每个节点随机生成一个投影方向，将当前节点中的所有样本投影到该方向上，再依据投影值对样本进行二分划分，并递归构建左右子树。

RP-Tree 假设当前节点包含样本 $S=\{\mathbf{x}_1,\mathbf{x}_2,\cdots,\mathbf{x}_n\}$，首先随机生成一个单位方向向量

$$\mathbf{w} \in \mathbb{R}^D, \qquad \|\mathbf{w}\|_2 = 1. \tag{18.36}$$

随后，将所有样本投影到该方向上：

$$p_i = \mathbf{x}_i^\top \mathbf{w}, \qquad i = 1, \cdots, n. \tag{18.37}$$

最后，根据所有投影值的中位数

$$p_{\text{median}} = \text{median}(p_1, p_2, \cdots, p_n) \tag{18.38}$$

构造划分超平面：

$$\mathbf{x}^\top \mathbf{w} = p_{\text{median}} \tag{18.39}$$

满足

$$\mathbf{x}_i^\top \mathbf{w} < p_{\text{median}} \tag{18.40}$$

的样本进入左子树，其余样本进入右子树。随后继续对子节点重复上述过程，直到叶节点满足停止划分条件。

因此，与 KD 树每次沿坐标轴进行划分不同，RP-Tree 每一次划分对应的是一个随机方向上的超平面。由于不同节点采用不同的随机方向，因此树结构能够更加灵活地适应高维空间中数据真实的分布形态，从而降低固定坐标轴划分带来的偏置。

18.2.3.0.1 Johnson–Lindenstrauss 引理 随机投影树之所以能够在随机选择划分方向之后仍然保持较好的检索性能，其理论基础来源于著名的 Johnson–Lindenstrauss 引理（简称 JL 引理）。JL 引理揭示了一个十分反直觉但又极其重要的事实：

性质 对于有限个高维数据点，只需要投影到一个远低于原始维度的随机低维空间，就能够以极高概率保持所有点对之间的距离关系。这也是随机投影能够广泛应用于高维检索、近似最近邻搜索以及随机投影树的重要理论依据。

Johnson 和 Lindenstrauss 最早在 1984 年的论文《Extensions of Lipschitz Mappings into a Hilbert Space》[5] 中研究了有限点集的低失真嵌入问题，并证明任意有限点集均可以嵌入到维度仅为 $O(\log n)$ 的低维 Hilbert 空间中，同时保持较小的距离失真。原论文中关于 Lemma 1 的表述如下所示：

引理 18.1 (Johnson–Lindenstrauss 原始引理)

对任意满足 $0 < \varepsilon < 1$ 的常数 ε，存在仅依赖 ε 的正常数 $K = K(\varepsilon)$；若集合 $A \subset \ell_2^n$ 且集合元素数量 $|A| = n (n = 2, 3, \ldots)$，则一定存在映射 f，将 A 映至 k 维欧氏空间 ℓ_2^k 的某个子集（其中维度 $k = \lceil K \log n \rceil$），并且该映射满足：$\|f\|_{\text{Lip}} \cdot \|f^{-1}\|_{\text{Lip}} \leq \frac{1+\varepsilon}{1-\varepsilon}$。

其中，ℓ_2^n 是 n 维欧几里得空间，即 2-范数空间；$|A| = n$ 表示集合 A 中一共含有 n 个向量（点）；$\lceil \cdot \rceil$ 是向上取整符号，代表向上取整数；$\|f\|_{\text{Lip}}$ 是映射 f 的 Lipschitz 常数；f^{-1} 表示映射 f 的逆映射，$\|f^{-1}\|_{\text{Lip}}$ 为逆映射的 Lipschitz 常数；$\dfrac{1+\varepsilon}{1-\varepsilon}$ 为映射整体的距离失真上界。

该引理在原文中的证明思路为：首先随机从正交群 $O(n)$ 抽取旋转矩阵 U，做随机 k 维正交投影 $f(x) = QUx$（Q 是取前 k 维坐标的投影算子）。然后用旋转不变 Haar 测度 + Lévy/Khintchine 集中不等式，证明：随机取的旋转 U 有极高概率，让集合 A 内所有归一化差向量 $y = \frac{x-y}{\|x-y\|}$ 经过投影后，半范数 $r(Uy)$ 落在狭窄区间 $[M_r(1-\tau), M_r(1+\tau)]$；取足够大 $k = K \log n$，让该概率严格大于 0 → 必然存在至少一个满足条件的旋转 U，对应的投影 f 就是满足双 Lipschitz 失真 $\frac{1+\tau}{1-\tau}$ 的低维嵌入映射；最后用体积填充论证说明 $k = O(\log n)$ 是不可改进的下界。

在 JL 原始论文后续的研究中，研究者们在此基础上逐渐发展出如今机器学习领域广泛使用的 JL 引理，其经典形式如下：

引理 18.2 (Johnson–Lindenstrauss 引理)

对于任意 $0 < \epsilon < 1$，以及 $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n \in \mathbb{R}^D$，存在映射 $f : \mathbb{R}^D \to \mathbb{R}^k$，其中 $k = \mathcal{O}(\frac{\log n}{\epsilon^2})$，使得

$$(1-\epsilon)\|\mathbf{x}_i - \mathbf{x}_j\|^2 \leq \|f(\mathbf{x}_i) - f(\mathbf{x}_j)\|^2 \leq (1+\epsilon)\|\mathbf{x}_i - \mathbf{x}_j\|^2 \tag{18.41}$$

这个现代版的 JL 引理证明核心会用到 Cramér-Chernoff 方法和单位模引理。其中 Cramér-Chernoff 方法的底层原理是对任意随机变量 Z、$\lambda > 0$ 有：

$$\mathbb{P}(Z \geq a) = \mathbb{P}(e^{\lambda Z} \geq e^{\lambda a}) \leq e^{-\lambda a}\mathbb{E}[e^{\lambda Z}] \tag{18.42}$$

对右侧关于 λ 求极小，得到最优指数衰减界，属于大偏差通用框架，专门处理独立随机变量求和（高斯平方和即卡方分布），即：

$$\mathbb{P}(Z \geq a) \leq \min_{\lambda > 0} e^{-\lambda a}\mathbb{E}[e^{\lambda Z}] \tag{18.43}$$

需要注意的是，Cramér-Chernoff 方法通常可以由马尔科夫不等式，或者切比雪夫不等式推导得出。而单位模引理的描述如下：

引理 18.3 (单位模引理)

设 $\mathbf{x}$ 为固定单位向量，随机矩阵 $\mathbf{R}$ 的元素独立重复采样自高斯分布 $\mathcal{N}(0, 1/d)$，$\epsilon \in (0,1)$ 是给定的常数。令 $Z = \|\mathbf{R}\mathbf{x}\|^2$，则有：

$$\mathbb{P}(\|Z - 1\| \geq \varepsilon) \leq 2\exp\left(-\frac{d\varepsilon^2}{8}\right) \tag{18.44}$$

因此，对于采样自高斯分布的随机矩阵构成映射 f，我们可以将 JL 引理写成：

引理 18.4 (Johnson–Lindenstrauss 引理)

对于任意 $0 < \epsilon < 1$，$\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n \in \mathbb{R}^D$，存在 $k > \frac{24 \log n}{\epsilon^2}$，随机矩阵 $\mathbf{R} \in \mathbb{R}^{k \times D}$ 独立重复采样自高斯分布 $\mathcal{N}(0, \frac{1}{k})$，那么至少有 $1 - \frac{1}{n}$ 的概率使得对于所有的 $i \neq j$，有下式成立：

$$(1-\epsilon)\|\mathbf{x}_i - \mathbf{x}_j\|^2 \leq \|\mathbf{R}\mathbf{x}_i - \mathbf{R}\mathbf{x}_j\|^2 \leq (1+\epsilon)\|\mathbf{x}_i - \mathbf{x}_j\|^2 \tag{18.45}$$

关于使用单位模引理对 JL 引理的数学证明可以查看参考资料 [9]。目前，Johnson–Lindenstrauss 引理有两种主流证明：1984 年原始论文中基于泛函分析视角的证明，以及适用于工程计算的高斯随机矩阵现代证明。二者核心论证框架一致，均采用随机存在性思路：先构造随机线性降维映射，利用集中不等式得到单点距离失真的指数衰减界，再通过并界约束全部点对，选取维度 $k = O(\log n/\varepsilon^2)$ 使合格映射存在，最终得到同阶低维维度下界。二者区别主要体现在随机变换与概率工具上。原始证明采用 Haar 均匀随机正交矩阵做空间旋转后截断投影，矩阵元素相互耦合，依靠球面 Lévy 不等式、Khintchine 不等式完成分析，结论以双 Lipschitz 常数乘积刻画失真，仅证明映射存在，无显式构造；现代证明使用元素独立的高斯随机矩阵，依托 Cramér–Chernoff 大偏差与单位模引理推导，直接给出平方距离扰动不等式，形式简洁且可直接编程实现，是算法与机器学习领域的通用版本。

JL 引理最令人惊讶的地方在于，其所需的目标维数几乎与原始维度 D 无关，而仅与样本数量 n 的对数成正比。举例来说，即使原始向量维度达到 $D = 10000$，若数据集中仅包含 $n = 1000$ 个样本，那么理论上仅需约两百维左右的随机投影空间，便能够以极高概率保持全部样本之间的两两距离。这也是随机投影能够广泛应用于高维近似最近邻搜索、随机投影树以及现代向量检索算法的重要理论基础。

18.2.3.0.2 随机投影树建树 随机投影树（Random Projection Tree，RP-Tree）仍然采用二叉树递归划分的数据组织方式，不同之处在于，每一次划分不再沿固定坐标轴进行，而是随机生成一个方向，并利用该方向对应的随机超平面完成空间划分。由于随机投影能够较好地保持样本之间的相对距离，因此树结构的划分效率主要取

决于数据的内在维数（Intrinsic Dimension），而不受原始空间维度的影响。

下面仍以前文 KD 树中的二维数据集为例说明随机投影树的构建过程，如图 18.6所示。首先随机生成一个方向向量，其对应的分割超平面恰好经过样本点 $C(80,15)$，于是整个二维空间被划分为两个区域。随后分别对子区域继续随机生成新的划分方向，递归完成空间划分。当某一区域中的样本数量低于预设阈值（例如 1 或 2）时，递归终止，该区域对应随机投影树的一个叶节点。

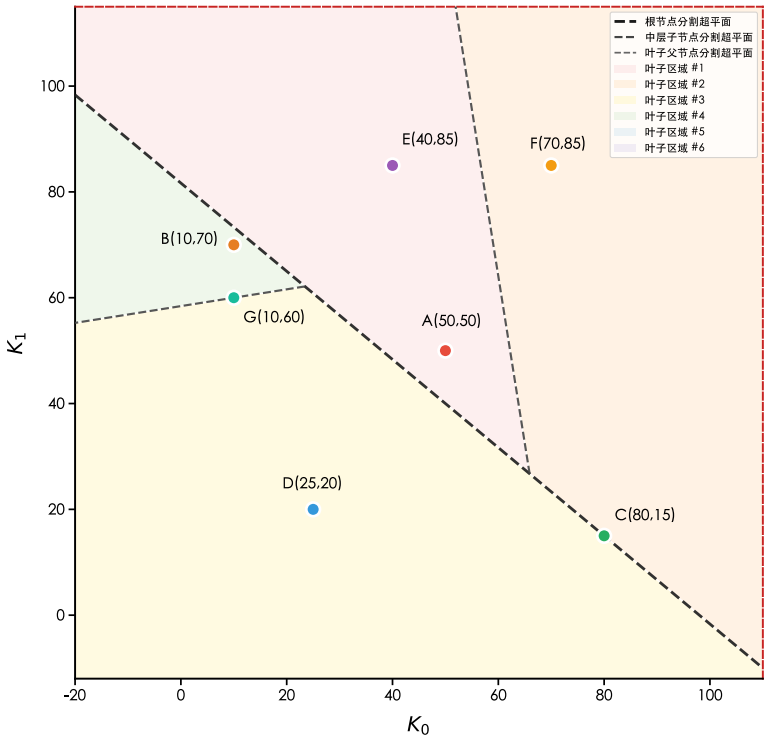

图 18.6: 二维空间中随机投影树划分示例。

在原始论文中，作者提出了两种主要的 RP-Tree 构建方式：RP-Tree-Max 和 RP-Tree-Mean。其中，RP-Tree-Max 的核心目标是不断减小当前节点包含数据点集合的最大直径（Maximum Diameter）。假设当前节点包含的数据集合为：$S = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n\}$，其构建步骤如下：

1. 随机生成单位方向向量 $\mathbf{v} \in \mathbb{R}^D$；
2. 从集合 S 中任选一个点 $\mathbf{x}$，再找到距离 $\mathbf{x}$ 最远的数据点 $\mathbf{y}$；
3. 计算当前节点的大致尺度 $|\mathbf{x} - \mathbf{y}|$；
4. 在随机方向 $\mathbf{v}$ 上计算所有样本投影：$p_i = \mathbf{x}_i^\top \mathbf{v}$；
5. 以投影中位数附近的随机扰动位置作为划分边界：

$$t = \text{median}(p_i) + \delta \tag{18.46}$$

其中

$$\delta \sim \text{Uniform}\left(-\frac{6|\mathbf{x}-\mathbf{y}|}{\sqrt{D}}, \frac{6|\mathbf{x}-\mathbf{y}|}{\sqrt{D}}\right) \tag{18.47}$$

6. 根据 $\mathbf{x}_i^\top \mathbf{v} \le t$ 将数据划分到左右子树。

在 RP-Tree-Max 中，相比于简单地使用投影中位数进行划分，随机扰动 δ 的引入可以避免数据分布特殊情况下出现退化划分，从而增强树结构的平衡性。需要注意的是，不管是 RP-Tree-Max 中的随机投影向量还是中位数的随机扰动，实际上都是理论推导中的必备要素，而不是可任意去掉的部分。其中中位数的随机扰动核心是用来控制破坏局部簇的坏分裂概率。

相比起 RP-Tree-Max 重点关注最大直径，RP-Tree-Mean 则进一步考虑数据的整体分布特征。首先定义当前

节点的数据直径（Diameter）：

$$\max_{\mathbf{x},\mathbf{y}\in S} |\mathbf{x}-\mathbf{y}| \tag{18.48}$$

以及平均直径（Average Diameter）：

$$\frac{1}{|S|^2}\sum_{\mathbf{x},\mathbf{y}\in S} |\mathbf{x}-\mathbf{y}|^2 \tag{18.49}$$

这里 $\Delta(S)$ 描述了最远两个样本之间的距离，$\Delta_A(S)$ 则反映整体数据的平均离散程度。

首先定义当前节点的数据直径（Diameter）：

$$\Delta(S) = \max_{\mathbf{x},\mathbf{y}\in S} \|\mathbf{x}-\mathbf{y}\| \tag{18.50}$$

以及平均直径（Average Diameter）：

$$\Delta_A^2(S) = \frac{1}{|S|^2}\sum_{\mathbf{x},\mathbf{y}\in S} \|\mathbf{x}-\mathbf{y}\|^2 \tag{18.51}$$

其中 $\Delta(S)$ 表示当前节点中最远两个样本之间的距离，$\Delta_A(S)$ 则刻画样本整体的平均离散程度。RP-Tree-Mean 会根据两者之间的关系动态选择划分策略：

- 若 $\Delta^2(S) \le c\cdot\Delta_A^2(S)$，则说明数据分布较为均匀，此时采用与 RP-Tree-Mean 完全相同的随机投影划分；
- 若 $\Delta^2(S) > c\cdot\Delta_A^2(S)$，则说明数据中可能存在离群点或者明显的长尾结构，此时采用基于均值距离的划分。

具体来说当 $\Delta^2(S) > c\cdot\Delta_A^2(S)$ 时，RP-Tree-Mean 的划分方式如下所示：

1. 计算当前节点下的样本均值：

$$\boldsymbol{\mu} = \frac{1}{|S|}\sum_{\mathbf{x}\in S}\mathbf{x} \tag{18.52}$$

2. 对每个样本计算其到均值的距离：

$$d_i = \|\mathbf{x}_i - \boldsymbol{\mu}\| \tag{18.53}$$

3. 计算距离的中位数：

$$d_{\text{med}} = \text{median}(d_1, d_2, \cdots, d_n) \tag{18.54}$$

4. 根据 $\|\mathbf{x}-\boldsymbol{\mu}\| \le d_{\text{med}}$ 将数据划分为两个子集。

与 RP-Tree-Max 相比，RP-Tree-Mean 能够根据当前节点的数据分布自适应选择划分策略。当数据呈现明显簇结构或者存在离群点时，基于均值距离的划分往往能够更快地缩小节点直径，从而提高树结构的质量。

总体来看，RP-Tree-Max 始终围绕减小节点最大直径展开，其理论分析较为直接；RP-Tree-Mean 则进一步利用数据整体分布信息，根据节点形状动态切换划分策略，因此能够获得更快的单元格收缩速度。论文证明，对于具有较低内在维数的数据，RP-Tree 的收敛速度主要由内在维数决定，而不依赖于原始空间维度，这也是随机投影树能够克服传统 KD 树维度灾难的重要原因。

18.2.3.0.3 随机投影树查询 随机投影树的查询过程与 KD 树类似，均采用自顶向下的递归遍历方式。对于给定查询向量 $\mathbf{q}$，首先从根节点开始，根据当前节点保存的随机划分规则判断查询点位于超平面的哪一侧，然后进入对应子树继续搜索。不断重复这一过程，直至到达叶节点。

设当前节点对应随机方向为 $\mathbf{v}$，划分阈值为 t，则查询阶段只需计算

$$s = \mathbf{q}^\top\mathbf{v} \tag{18.55}$$

随后根据

$$s \le t \tag{18.56}$$

决定进入左子树，否则进入右子树。到达叶节点之后，对叶节点内保存的样本进行精确距离计算即可得到最近邻结果。

与 KD 树不同，RP-Tree 论文主要关注树结构的理论性质，因此并未设计复杂的回溯搜索策略，而是依赖随机划分带来的空间收缩性质保证查询质量。

18.2.3.0.4 Annoy 中的随机投影树实现 Spotify 开源的 Annoy（Approximate Nearest Neighbors Oh Yeah）是一种基于多棵随机投影树（Random Projection Trees）的近似最近邻检索库，具有支持内存映射（Memory Mapping，mmap）、索引构建速度快、查询效率高、实现简单以及易于分布式部署等特点，因此在中小规模向量检索和推荐系统中得到了广泛应用。

需要注意的是，Annoy 虽然同样采用随机投影树作为底层索引结构，但其实现方式与 RP-Tree 原始论文并不完全一致。RP-Tree 更加关注随机划分所对应的理论性质，希望证明树结构的收敛速度仅依赖于数据的内在维数；而 Annoy 则更加关注工程效率，因此采用了一系列更加适合工业实践的优化策略，例如 Two-Means 划分、多棵随机树索引以及 Best-First 搜索等。

Annoy 中每个内部节点对应一个随机超平面。为了构造该超平面，Annoy 并没有直接采用 RP-Tree 中的随机投影方式，而是首先利用 Two-Means 算法寻找两个具有代表性的中心点，然后利用两个中心点构造划分超平面。Two-Means 算法的实现如代码18.1所示。

代码 18.1: Annoy 库中的二均值法代码。

```
template<typename T, typename Random, typename Distance, typename Node>
inline void two_means(const vector<Node*>& nodes, int f, Random& random, bool cosine, Node* p, ↩
    ↪ Node* q) {
  static int iteration_steps = 200;
  size_t count = nodes.size();

  size_t i = random.index(count);
  size_t j = random.index(count-1);
  j += (j >= i); // ensure that i != j

  Distance::template copy_node<T, Node>(p, nodes[i], f);
  Distance::template copy_node<T, Node>(q, nodes[j], f);

  if (cosine) {
    Distance::template normalize<T, Node>(p, f); Distance::template normalize<T, Node>(q, f);
  }
  Distance::init_node(p, f);
  Distance::init_node(q, f);

  int ic = 1, jc = 1;
  for (int l = 0; l < iteration_steps; l++) {
    size_t k = random.index(count);
    T di = ic * Distance::distance(p, nodes[k], f),
      dj = jc * Distance::distance(q, nodes[k], f);
    T norm = cosine ? Distance::template get_norm<T, Node>(nodes[k], f) : 1;
    if (!(norm > T(0))) {
      continue;
    }
    if (di < dj) {
      Distance::update_mean(p, nodes[k], norm, ic, f);
      Distance::init_node(p, f);
      ic++;
```

```
    } else if (dj < di) {
      Distance::update_mean(q, nodes[k], norm, jc, f);
      Distance::init_node(q, f);
      jc++;
    }
  }
}
```

整个 Two-Means 算法可以概括为如下几个步骤：

1. 首先从当前节点对应的数据集中随机选择两个不同的数据点作为初始中心点 p 和 q；
2. 在迭代过程中，每次随机采样一个数据点 k；
3. 分别计算该数据点到两个中心点的距离，并根据距离远近将其分配给距离更近的一侧；
4. 利用增量平均（Incremental Mean）的方式实时更新对应中心的位置；
5. 重复上述过程若干轮之后，最终得到两个能够较好代表当前数据分布的中心点 p 和 q。

需要注意的是，Two-Means 并不是传统 K-Means 聚类算法。它并不会遍历全部样本进行多轮 EM 优化，而是采用随机采样结合在线均值更新的方法不断修正两个聚类中心，因此计算复杂度远低于标准 K-Means，更适合作为随机投影树建树阶段的快速划分算法。

得到两个中心点之后，Annoy 在上层函数 create_split 中构造真正的随机划分超平面，其核心代码如下所示：

代码 18.2: Annoy 库中的节点切分代码。

```
template<typename S, typename T, typename Random>
static inline void create_split(const vector<Node<S, T>*>& nodes, int f, size_t s, Random& random,↩
    ↪  Node<S, T>* n) {
   Node<S, T>* p = (Node<S, T>*)alloca(s);
   Node<S, T>* q = (Node<S, T>*)alloca(s);
   two_means<T, Random, Euclidean, Node<S, T> >(nodes, f, random, false, p, q);
   for (int z = 0; z < f; z++)
      n->v[z] = p->v[z] - q->v[z];
   Base::normalize<T, Node<S, T> >(n, f);
   n->a = 0.0;
   for (int z = 0; z < f; z++)
      n->a += -n->v[z] * (p->v[z] + q->v[z]) / 2;
}
```

在 create_split 函数中，首先调用 Two-Means 获得两个中心点 p 和 q，随后计算二者连线方向

$$\mathbf{n} = \mathbf{pq} \tag{18.57}$$

并进行归一化处理：

$$\mathbf{n} \leftarrow \frac{\mathbf{p} - \mathbf{q}}{\|\mathbf{p} - \mathbf{q}\|} \tag{18.58}$$

该向量即作为划分超平面的法向量。随后再计算超平面的偏置项

$$a = \mathbf{n}^{\mathrm{T}} \frac{\mathbf{p} + \mathbf{q}}{2} \tag{18.59}$$

因此最终划分超平面可以写成

$$\mathbf{n}^{\mathrm{T}} x + a = 0 \tag{18.60}$$

对于任意样本 $\mathbf{x}$，若

$$\mathbf{n}^{\mathrm{T}} x + a < 0 \tag{18.61}$$

则进入左子树；否则进入右子树。

可以看出，Annoy 最终采用的是” 聚类中心 + 随机初始化” 的划分方式，而并非 RP-Tree 论文中的随机方向加中位数扰动划分。因此，Annoy 虽然继承了随机投影树的整体思想，但更加偏向一种工程化实现，在实际数据集上通常能够获得更好的划分质量。

在实际应用中，为进一步提高召回率，Annoy 通常不会仅建立一棵随机投影树，而是同时建立多棵相互独立的随机投影树：

$$T_1, T_2, \cdots, T_M \tag{18.62}$$

每棵树由于随机初始化不同，因此对应不同的空间划分方式。查询时 Annoy 分别在所有树中检索，将多个叶节点中的候选集合进行合并，再执行精确距离排序获得最终结果。一般来说，树的数量越多，检索召回率（Recall）通常越高，但与此同时索引大小、内存占用以及查询时间 (Latency) 也会相应增加。因此实际工程中通常通过调节树的数量实现 Recall 与 Latency 之间的权衡。

在 Annoy 的查询实现中，与传统随机投影树逐棵树独立搜索不同，Annoy 采用了一种全局 Best-First Search 策略。其核心思想是不再固定完成一棵树的搜索之后再进入下一棵树，而是统一维护一个优先队列（Priority Queue），动态决定当前最值得继续扩展的节点。

对应实现代码如代码18.3所示。

代码 18.3: Annoy 库中的随机投影树查询。

```
void _get_all_nns(const T* v, size_t n, int search_k, vector<S>* result, vector<T>* distances) ↩
    ↪ const {
  Node* v_node = (Node *)alloca(_s);
  D::template zero_value<Node>(v_node);
  memcpy(v_node->v, v, sizeof(T) * _f);
  D::init_node(v_node, _f);

  std::priority_queue<pair<T, S> > q;

  if (search_k == -1) {
    search_k = n * _roots.size();
  }

  for (size_t i = 0; i < _roots.size(); i++) {
    q.push(make_pair(Distance::template pq_initial_value<T>(), _roots[i]));
  }

  std::vector<S> nns;
  while (nns.size() < (size_t)search_k && !q.empty()) {
    const pair<T, S>& top = q.top();
    T d = top.first;
    S i = top.second;
    Node* nd = _get(i);
    q.pop();
    if (nd->n_descendants == 1 && i < _n_items) {
      nns.push_back(i);
    } else if (nd->n_descendants <= _K) {
      const S* dst = nd->children;
      nns.insert(nns.end(), dst, &dst[nd->n_descendants]);
    } else {
```

```
      T margin = D::margin(nd, v, _f);
      q.push(make_pair(D::pq_distance(d, margin, 1), static_cast<S>(nd->children[1])));
      q.push(make_pair(D::pq_distance(d, margin, 0), static_cast<S>(nd->children[0])));
    }
  }

  // Get distances for all items
  // To avoid calculating distance multiple times for any items, sort by id
  std::sort(nns.begin(), nns.end());
  vector<pair<T, S> > nns_dist;
  S last = -1;
  for (size_t i = 0; i < nns.size(); i++) {
    S j = nns[i];
    if (j == last)
      continue;
    last = j;
    if (_get(j)->n_descendants == 1) // This is only to guard a really obscure case, #284
      nns_dist.push_back(make_pair(D::distance(v_node, _get(j), _f), j));
  }

  size_t m = nns_dist.size();
  size_t p = n < m ? n : m; // Return this many items
  std::partial_sort(nns_dist.begin(), nns_dist.begin() + p, nns_dist.end());
  for (size_t i = 0; i < p; i++) {
    if (distances)
      distances->push_back(D::normalized_distance(nns_dist[i].first));
    result->push_back(nns_dist[i].second);
  }
}
```

总结起来，Annoy 中随机投影树的整个查询流程可以概括为以下几个步骤：

1. 首先将所有随机投影树的根节点加入优先队列，每个队列元素均保存一个 $(priority, node)$ 二元组；
2. 每次从优先队列中弹出优先级最高的节点进行扩展；
3. 若当前节点对应单个样本，则直接加入候选集合；
4. 若当前节点已经属于叶子桶（Bucket），则直接将桶内全部样本加入候选集合；
5. 若当前节点仍属于内部节点，则首先计算查询向量与当前划分超平面的 Margin：

$$margin = \mathbf{n}^{\mathrm{T}}\mathbf{q} + a \tag{18.63}$$

   随后分别计算左右子节点对应的优先值，并重新压入优先队列。
6. 重复上述过程，直到候选样本数量达到参数 search_k 或者优先队列为空。

在得到候选集合之后，Annoy 并不会直接返回结果，而是首先按照样本 ID 排序去重，然后重新计算查询向量与所有候选样本之间的精确距离，最后利用 partial_sort 的方式取得距离最近的 Top-p 个样本作为最终检索结果。与逐棵树深度优先遍历相比，Best-First Search 这种基于 margin 优先值启发式的搜索方法能够根据当前节点与查询向量之间的 Margin 动态调整搜索顺序，使有限的搜索预算优先分配给更有可能包含最近邻的子树。因此，在相同搜索代价下通常能够获得更高的 Recall。

Annoy 中的另一个重要参数是 search_k，它表示整个查询过程中允许访问的最大候选节点数。当增大 search_k 时，优先队列能够扩展更多节点，因此 Recall 通常会进一步提高，但查询延迟也会相应增加；反之，减小 search_k 能够获得更快的查询速度，但可能导致部分真正的近邻尚未被访问。因此，在实际工程中通常同时调节随机投

影树数量 (n_trees) 以及搜索预算 (search_k)，从而实现 Recall、Latency 以及内存占用之间的综合权衡。

18.3 局部敏感哈希 (LSH)

局部敏感哈希 (Locality Sensitive Hashing, LSH) 是哈希函数的一种，通常哈希函数只是为了高效地将数据映射成哈希值，同时尽量减少数据映射之后的哈希值冲突问题。传统哈希算法也无法保证相似的数据在映射之后仍然能够具有相同的哈希值或者哈希分桶。而 LSH 方法是一种特殊的哈希方法，在将数据映射成哈希值的同时还能保证局部相似数据其哈希值也有一定的相似性。用数学形式化的语言来描述，LSH 是一类定义在哈希函数族 $\mathcal{F}$ 上的概率分布。对于任意两个对象 x 和 y，若随机从哈希函数族 $\mathcal{F}$ 中选取一个哈希函数 h，则满足：

$$P_{h\in\mathcal{F}}[h(x)=h(y)]=sim(x,y) \tag{18.64}$$

其中，$sim(x,y)\in[0,1]$ 表示对象 x 与对象 y 之间定义好的相似度函数。这样的哈希方案能够将原始对象压缩为紧凑的哈希表示（Sketch）。借助这些紧凑表示，不仅可以快速估计两个对象之间的相似度，还能够设计出高效的近似最近邻（Approximate Nearest Neighbor，ANN）搜索算法以及聚类算法。

下面我们重点对一些 LSH 的哈希族进行介绍，包括了 MinHash、SimHash 这些哈希方法，主要优化 Jaccard 的集合相似度。还有基于随机投影的哈希族，主要用于优化欧式距离和余弦相似度距离。

18.3.1 MinHash

对于集合型数据而言，最常用的相似度度量之一是 Jaccard 相似系数。例如，两篇文档可以表示为词项（Token）集合，两位用户可以表示为点击商品集合，一个网页可以表示为包含的 URL 集合。在这些场景中，我们通常更关心两个集合重叠程度，而不是向量之间的欧氏距离。

设两个集合分别为 A 和 B，其 Jaccard 相似度定义为：

$$J(A,B)=\frac{|A\cap B|}{|A\cup B|} \tag{18.65}$$

其中，$|A\cap B|$ 表示两个集合公共元素数量，$|A\cup B|$ 表示两个集合所有不同元素数量。Jaccard 相似度越接近 1，说明两个集合越相似；越接近 0，则说明两个集合几乎没有公共元素。

然而，对于大规模集合数据而言，直接计算交并集代价较高。例如推荐系统中，每个用户可能对应几千甚至几万个点击 Item，直接计算 $|A\cap B|$ 和 $|A\cup B|$ 都需要遍历整个集合，因此难以应用于大规模 ANN 检索。MinHash（Minimum Hashing）正是针对这一问题提出的一种局部敏感哈希方法，其核心思想十分简单：

定义 18.6

利用随机排列，仅保留集合中第一个出现的元素作为集合的哈希值，使两个集合发生哈希碰撞的概率恰好等于它们的 Jaccard 相似度。

因此，MinHash 可以利用多个随机哈希值近似估计 Jaccard 相似度，而无需真正计算集合交并集。下面通过图18.7介绍 MinHash 签名（Signature）的生成过程。假设共有 n 篇文档，全部 Token 构成大小为 d 的词典，则可以得到如下二值特征矩阵：

$$\mathbf{X}\in\{0,1\}^{d\times n} \tag{18.66}$$

其中每一列表示一篇文档，每一行对应一个 Token，元素为 1 表示当前文档包含对应 Token，否则为 0。

MinHash 首先随机生成一个元素索引的排列（Permutation）：

$$\pi:\{1,\cdots,d\}\to\{1,\cdots,d\} \tag{18.67}$$

例如图中的第一组随机排列可写为：

$$\pi=\{2,5,8,3,6,1,7,4\} \tag{18.68}$$

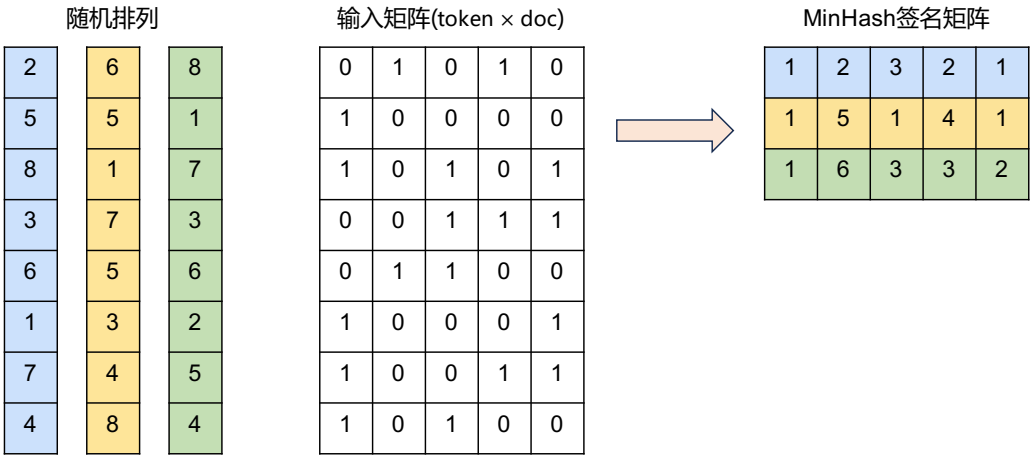

图 18.7: MinHash 生成签名矩阵示例。

表示原始第 1 行移动到第 2 行，第 2 行移动到第 5 行，以此类推。

对于任意文档（即矩阵的一列），只需要找到排列以后第一个出现 1 的位置即可作为该随机排列对应的哈希值，即

$$h_\pi(C) = \min_{r \in C} \pi(r) \tag{18.69}$$

其中 C 表示当前文档中所有取值为 1 的行索引集合，$\pi(r)$ 表示随机排列后的索引位置。由于一次随机排列只能产生一个哈希值，所以在实际应用中通常会独立生成 m 组随机排列：

$$\pi_1, \pi_2, \cdots, \pi_m \tag{18.70}$$

从而得到长度为 m 的 MinHash 签名：

$$(h_{\pi_1}, h_{\pi_2}, \cdots, h_{\pi_m}) \tag{18.71}$$

这样，所有文档组合起来便形成图18.7右侧所示的 MinHash 签名矩阵。

MinHash 作为 LSH 哈希族的重要一员，它所具备的最重要的性质如下：

性质 *对于任意两个集合 A 和 B，MinHash 碰撞概率满足*

$$P\left[h_\pi(A) = h_\pi(B)\right] = \frac{|A \cap B|}{|A \cup B|} = J(A, B) \tag{18.72}$$

该性质也是 MinHash 能够成为局部敏感哈希（LSH）的根本原因。证明过程实际上十分直观。设 $U = A \cup B$ 随机排列以后，集合 U 中的所有元素都有相同概率成为排列后的第一个元素。若排列后的第一个元素来自交集 $A \cap B$，则该元素同时属于两个集合，因此 $h_\pi(A) = h_\pi(B)$。反之，只要排列后的第一个元素来自 $A \setminus B$ 或 $B \setminus A$（即 A 或 B 集合独占的子集），两个集合得到的最小排列值必然不同，因此不会发生碰撞。

由于随机排列满足均匀分布，因此排列后第一个元素落入交集的概率正好等于交集元素数量占并集元素数量的比例，即

$$\frac{|A \cap B|}{|A \cup B|} \tag{18.73}$$

从而得到上述性质。

根据大数定律，如果独立构造 m 个 MinHash 函数，则两个集合的 MinHash 签名中相同位置所占比例

$$\hat{J} = \frac{1}{m} \sum_{i=1}^{m} \mathbf{1}\left(h_i(A) = h_i(B)\right) \tag{18.74}$$

将收敛到真实 Jaccard 相似度：

$$\hat{J} \to J(A, B) \tag{18.75}$$

因此，MinHash 签名长度越长，Jaccard 相似度估计方差越小，估计结果越稳定；但与此同时，存储空间和计算代价也会相应增加，因此实际工程中通常会在计算效率与估计精度之间进行权衡。按照原始随机排列的 MinHash

其签名长度与对 Jaccard 相似度的估计精度关系如下图18.8所示。

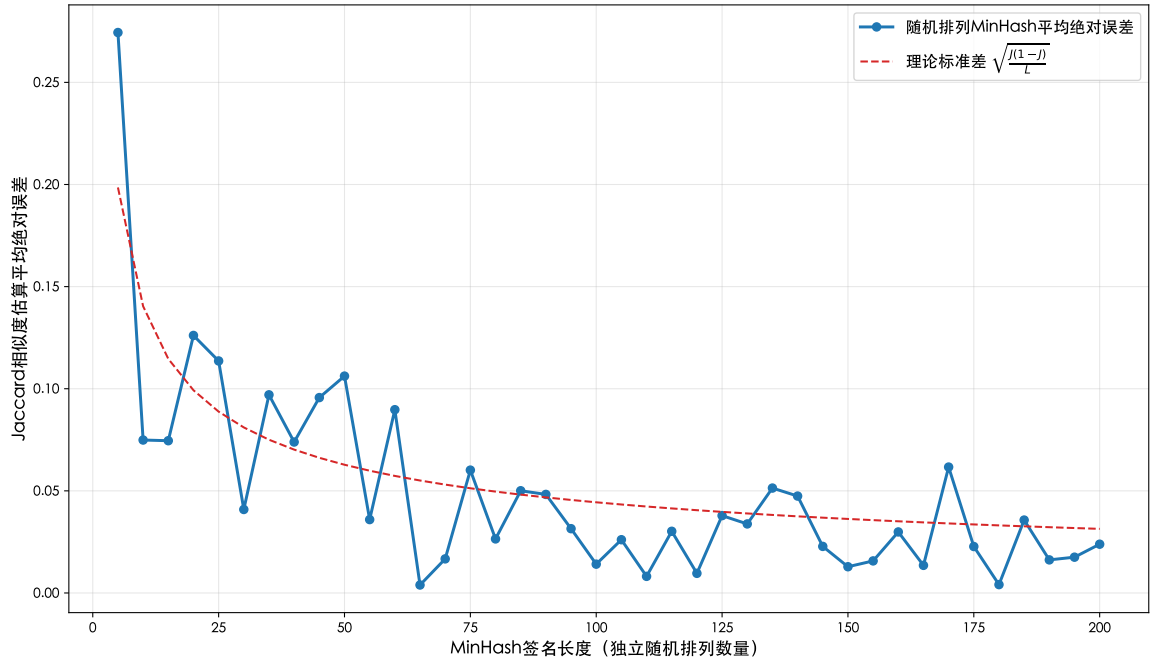

图 18.8: MinHash 签名长度与准确度关系。

从图中可以看出，随着随机排列数量的增加，即 MinHash 签名长度增加，误差会逐步下降。这里的理论预估标准差为

$$std = \sqrt{\frac{J(1-J)}{L}} \tag{18.76}$$

其中 J 表示真实的 Jaccard 相似度，L 表示 MinHash 签名的长度。我们根据 MinHash 的哈希碰撞概率等于 Jaccard 相似度的性质，可以定义 0-1 的随机变量：

$$X_i = \begin{cases} 1, & \text{sig}_i(A) = \text{sig}_i(B) \\ 0, & \text{sig}_i(A) \neq \text{sig}_i(B) \end{cases} \tag{18.77}$$

由于 X_i 服从伯努利分布，可以得到：

$$\mathbb{E}[X_i] = 1 \cdot J + 0 \cdot (1-J) = J \tag{18.78}$$

$$Var[X_i] = J(1-J)^2 + (1-J)(0-J)^2 = J(1-J) \tag{18.79}$$

而对于长度为 L 的 MinHash 签名，可以取估计值为：

$$\hat{J} = \frac{1}{L}\sum_{i=1}^{L} X_i \tag{18.80}$$

由于 L 个 X_i 相互独立，所以其方差为：

$$Var\left(\sum_{i=1}^{L} X_i\right) = \sum_{i=1}^{L} Var(X_i) = L \cdot J(1-J) \tag{18.81}$$

所以：

$$Var(\hat{J}) = Var\left(\frac{1}{L}\sum_{i=1}^{L} X_i\right) = \frac{1}{L^2} \cdot L \cdot J(1-J) = \frac{J(1-J)}{L} \tag{18.82}$$

最终得到理论标准差计算为 $std = \sqrt{\frac{J(1-J)}{L}}$。

下面给出原始 MinHash 基于随机排列的代码版本，如代码18.4所示：

代码 18.4: 基于随机排列的 MinHash 示例代码。

```
class MinHashPermutation:
    def __init__(self, num_hash, universe_size, seed=42):
        self.num_hash = num_hash
        self.universe_size = universe_size
        self.seed = seed
        random.seed(seed)
        # 预生成num_hash套独立全局随机排列
        self.permutations = []
        for _ in range(num_hash):
            perm = list(range(universe_size))
            random.shuffle(perm)
            # 构建反向映射: 元素x -> 置换后的位置rank
            rank = {x: idx for idx, x in enumerate(perm)}
            self.permutations.append(rank)

    def get_signature(self, s: set):
        sig = np.empty(self.num_hash, dtype=int)
        for i, rank_dict in enumerate(self.permutations):
            # 取集合内所有元素在该置换下的最小rank（理论minhash定义）
            min_rank = min(rank_dict[x] for x in s)
            sig[i] = min_rank
        return sig
```

但原始基于随机排列的 MinHash 方法在数据维度较高的时候通常在保存完整随机排列映射表时需要巨大的内存开销，同时生成高维全排列的耗时也会很高，不适合大规模落地。因此，工程上通常会采用线性哈希这种更加轻量级的计算方式来替代随机排列，将每个元素通过线性哈希映射到随机的取值，这样在哈希映射之后取值范围比较大的情况下碰撞概率比较低。最后取集合中哈希值最小的元素最为 MinHash 的签名，这样从理论上依然是等价的。具体如代码18.5所示：

代码 18.5: 基于线性哈希的 MinHash 示例代码。

```
class MinHash:
    def __init__(self, num_hash: int, seed=42):
        self.num_hash = num_hash
        self.seed = seed
        random.seed(seed)
        # 生成随机哈希系数 h(x) = (a*x + b) mod p
        self.p = 2**31 - 1 # 大素数
        self.a = [random.randint(1, self.p-1) for _ in range(num_hash)]
        self.b = [random.randint(0, self.p-1) for _ in range(num_hash)]

    def get_signature(self, s: set):
        """输入集合，返回minhash签名"""
        sig = [float("inf")] * self.num_hash
        for x in s:
            for i in range(self.num_hash):
                h = (self.a[i] * x + self.b[i]) % self.p
                if h < sig[i]:
```

```
            sig[i] = h
    return np.array(sig)
```

需要注意的是，MinHash 本身仅提供了一种紧凑的集合表示方式，并不能直接完成近似最近邻搜索。在实际应用中，通常还需要进一步结合 LSH Banding（Band 划分）技术，将多个 MinHash 签名划分为若干 Band，并利用 Band 内部完全相同的 Signature 作为哈希键，将可能相似的对象映射到同一个 Bucket 中，从而大幅减少候选集合规模，实现大规模集合数据的近似最近邻检索。后续我们将在 LSH Banding 部分继续介绍这一过程。

18.3.2 SimHash

MinHash 主要面向集合数据，通过随机排列估计 Jaccard 相似度。然而，在实际工程中，大量数据并不能自然表示为集合，而更适合表示为高维实值向量。例如，网页可以表示为 TF-IDF 特征向量，文本 Embedding、用户 Embedding 以及 Item Embedding 也都是高维连续向量。这类数据更关心的是向量之间的夹角，而不是集合交并关系，因此更加适合采用余弦相似度（Cosine Similarity）进行度量：

$$\mathrm{CosSim}(\mathbf{x},\mathbf{y}) = \frac{\mathbf{x}^T\mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} \tag{18.83}$$

18.3.2.0.1 基于随机超平面的 LSH 针对这一问题，Charikar 于 2002 年的经典论文《Similarity Estimation Techniques from Rounding》[2] 提出了 SimHash 算法。该论文首次建立了随机舍入（Randomized Rounding）、近似算法以及局部敏感哈希（LSH）之间的联系，并提出了适用于余弦相似度的随机超平面（Random Hyperplane）LSH 方案。如今所说的 SimHash，本质上就是随机超平面 LSH（Random Hyperplane LSH）。

SimHash 的核心思想非常简单：

定义 18.7

SimHash 利用随机超平面对向量空间进行划分，每一个随机超平面都对应哈希值中的一个 bit。假设输入向量为

$$\mathbf{x} = [x_1, x_2, \cdots, x_d]^T \tag{18.84}$$

首先随机生成一个 d 维随机向量

$$\mathbf{r} = [r_1, r_2, \cdots, r_d]^T, \qquad r_i \sim \mathcal{N}(0,1) \tag{18.85}$$

然后计算向量在随机方向上的投影

$$s = \mathbf{x}^T\mathbf{r}. \tag{18.86}$$

若投影位于超平面一侧，则输出 1；否则输出 0，因此对应的哈希函数可以写成：

$$h_{\mathbf{r}}(\mathbf{x}) = \mathrm{sign}(\mathbf{x}^T\mathbf{r}) = \begin{cases} 1, & \mathbf{x}^T\mathbf{r} \geq 0, \\ 0, & \mathbf{x}^T\mathbf{r} < 0. \end{cases} \tag{18.87}$$

♣

从几何上来看，随机向量 $\mathbf{r}$ 实际上对应一个经过原点的随机超平面，其法向量就是 $\mathbf{r}$。哈希值仅表示数据点位于该超平面的哪一侧，因此每个随机超平面只能提供 1 bit 信息。为了获得更稳定的表示，通常独立生成 L 个随机超平面：

$$\mathbf{r_1}, \mathbf{r_2}, \cdots, \mathbf{r_L} \tag{18.88}$$

分别计算对应的哈希值，最终得到长度为 L 的二进制签名（Fingerprint）：

$$\mathrm{Sig}(\mathbf{x}) = [h_{\mathbf{r}_1}(\mathbf{x}), h_{\mathbf{r}_2}(\mathbf{x}), \cdots, h_{\mathbf{r}_L}(\mathbf{x})] \tag{18.89}$$

随机超平面 LSH 具有如下最重要的性质：

性质 设两个向量之间夹角为 θ，则随机超平面将二者划分到同一侧的概率满足

$$\mathbb{P}\left[h(\mathbf{x}) = h(\mathbf{y})\right] = 1 - \frac{\theta}{\pi} \tag{18.90}$$

其中

$$\theta = \arccos\left(\frac{\mathbf{x}^T\mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}\right) \tag{18.91}$$

因此，两个向量越相似（夹角越小），产生相同哈希值的概率越大。

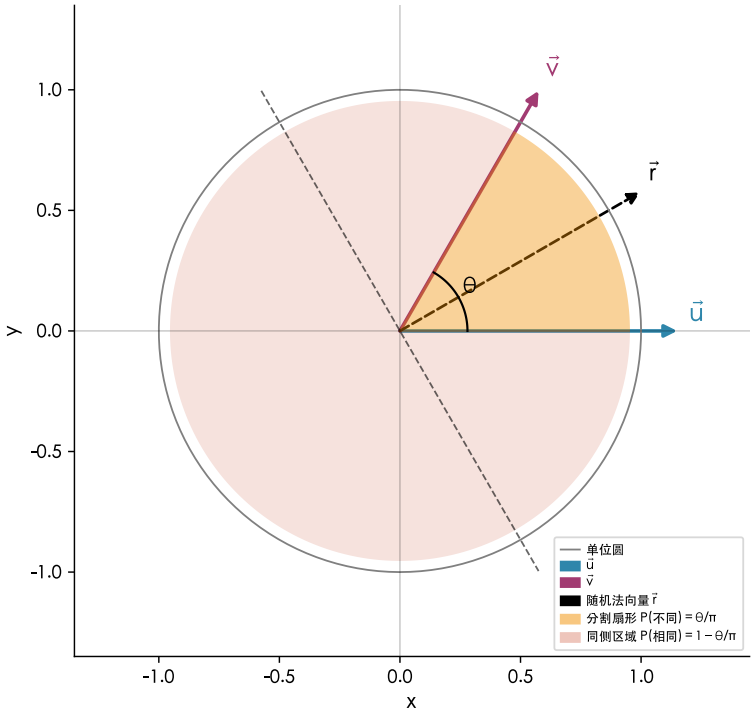

图 18.9: 随机超平面 LSH 几何解释：夹角与哈希碰撞概率。

这一性质具有非常直观的几何解释。如图18.9所示，对于两个夹角为 θ 的向量，只有当随机超平面的法向量落入夹角对应的扇形区域时，超平面才会将两个向量分到不同两侧。由于随机方向服从均匀分布，因此发生这种情况的概率正好为 θ/π，于是得到

$$P(\text{不同}) = \frac{\theta}{\pi} \tag{18.92}$$

因此

$$P(\text{相同}) = 1 - \frac{\theta}{\pi} \tag{18.93}$$

由于每一位哈希值都服从这一概率分布，因此长度为 L 的签名中，两条签名的汉明距离（Hamming Distance）可以近似估计原始向量夹角。设两条签名长度均为 L，汉明距离为 H，则有

$$\frac{H}{L} \approx \frac{\theta}{\pi} \tag{18.94}$$

进一步可得到

$$\theta \approx \pi\frac{H}{L} \tag{18.95}$$

从而可以利用汉明距离快速近似计算余弦相似度，而无需重新计算高维向量之间的内积。

然而，上述原始 SimHash 方法仍然存在两个明显的问题。首先，对于文本数据而言，其维度通常高达几十万甚至上百万。如果需要保存 L 个随机投影向量，则存储复杂度达到 $O(Ld)$；其次，每次计算哈希函数都需要完成一次 d 维内积运算，大量零元素参与计算，导致时间复杂度较高。因此，原始 SimHash 更适合作为一种理论构造，而不适合直接用于海量网页检索。

18.3.2.0.2 谷歌大规模 SimHash 为了使 SimHash 能够真正应用于工业搜索系统，Google 在 2007 年的论文《Detecting Near-Duplicates for Web Crawling》[7] 中，对 SimHash 进行了工程化实现，并成功应用于网页去重系统。需要注意的是，Google 并没有改变随机超平面 LSH 的理论基础，而是重新设计了随机向量的构造方式以及近重复文档的快速检索算法，使 SimHash 首次能够支持数十亿网页的近重复检测。

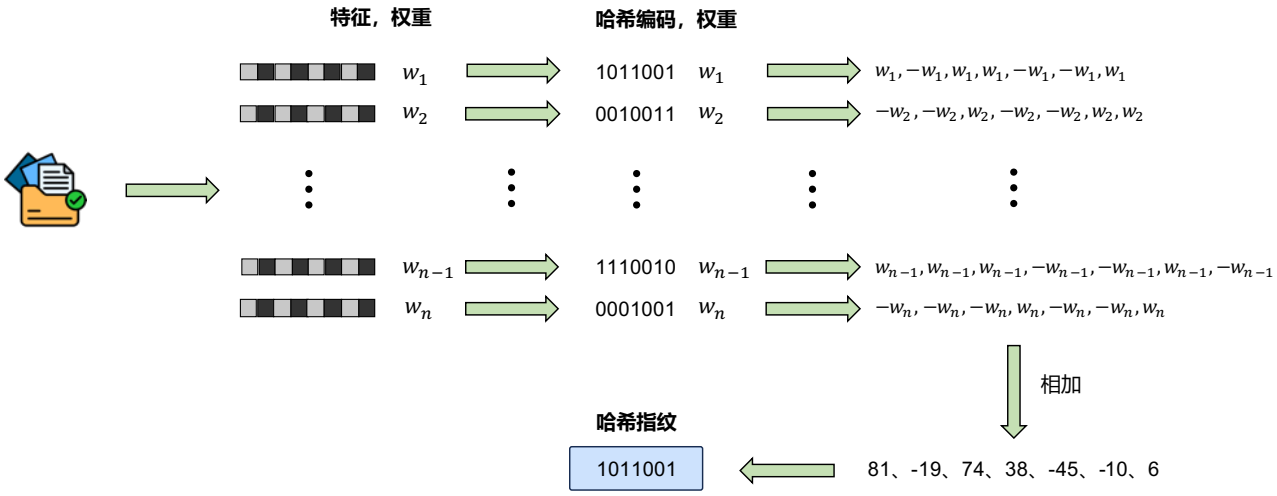

图 18.10: 谷歌 SimHash 计算流程。

谷歌 SimHash 整体流程如图18.10所示。首先，从文档中提取带权特征（例如 TF-IDF），然后利用普通哈希函数将每个特征映射为一个固定长度（通常 64 位）的二进制编码。随后，将编码中的每一位按照如下方式转换：

- 若当前 bit 为 1，则对应位置累加该特征权重；
- 若当前 bit 为 0，则对应位置减去该特征权重。

最终，对所有特征得到 64 维累计权重向量

$$[w_1, w_2, \cdots, w_{64}], \tag{18.96}$$

再依据每一维权重的正负生成最终指纹：

$$w_i \geq 0 \Rightarrow 1, \qquad w_i < 0 \Rightarrow 0 \tag{18.97}$$

整个过程无需显式生成高维随机投影矩阵，仅需维护固定长度的哈希编码即可完成随机投影的近似计算，大幅降低了存储和计算开销。

总体而言，MinHash 与 SimHash 分别对应两类不同的数据相似性问题。MinHash 主要面向集合数据，通过随机排列估计 Jaccard 相似度；SimHash 则主要面向实值向量，通过随机超平面近似保持余弦相似度，并利用汉明距离完成快速近邻检索。二者均属于局部敏感哈希（LSH）的经典代表算法，只是分别对应不同的相似度度量。

18.3.2.0.3 谷歌 SimHash 与随机超平面 LSH 的等价性 谷歌 SimHash 虽然在实现形式上与原始随机超平面 LSH 存在较大差异，但两者本质上是等价的随机投影过程。设文档包含 m 个特征，每个特征对应权重 w_i，其哈希函数生成的 L 位二进制编码记为

$$h(f_i) = [b_{i1}, b_{i2}, \cdots, b_{iL}] \tag{18.98}$$

其中 $b_{ij} \in \{0, 1\}$。谷歌 SimHash 首先将二进制编码转换为 $\{-1, +1\}$ 表示，即

$$r_{ij} = 2b_{ij} - 1, \tag{18.99}$$

于是每一个特征实际上对应一个长度为 L 的随机符号向量

$$\mathbf{r}_i = [r_{i1}, r_{i2}, \cdots, r_{iL}]^T. \tag{18.100}$$

由于普通哈希函数具有较好的随机性，因此可以认为各 bit 近似独立且服从

$$P(r_{ij} = 1) = P(r_{ij} = -1) = \frac{1}{2} \tag{18.101}$$

因此，每个特征对应的随机向量实际上由独立均匀分布的 $-1,+1$ 随机变量组成，与原始随机超平面 LSH 中的随机投影向量具有相同的随机性，因此可以看作随机投影向量的一种离散化实现。

随后谷歌 SimHash 计算

$$\mathbf{s}=\sum_{i=1}^{m} w_i \mathbf{r}_i \tag{18.102}$$

最后再根据各维符号生成最终指纹：

$$\text{SimHash}(\mathbf{x})=\text{sign}(\mathbf{s}) \tag{18.103}$$

另一方面，原始随机超平面 LSH 对于第 j 位哈希值计算方式为

$$h_j(\mathbf{x})=\text{sign}(\mathbf{r}_j^T\mathbf{x}) \tag{18.104}$$

其中 $\mathbf{r}_j$ 表示第 j 个随机超平面的法向量。若将文档表示为特征权重向量

$$\mathbf{x}=[w_1,w_2,\cdots,w_m]^T \tag{18.105}$$

并令第 j 位所有特征对应的随机符号组成随机向量

$$\mathbf{r}_j=[r_{1j},r_{2j},\cdots,r_{mj}]^T \tag{18.106}$$

则谷歌 SimHash 第 j 位计算结果满足

$$s_j=\sum_{i=1}^{m} w_i r_{ij}=\mathbf{r}_j^T\mathbf{x} \tag{18.107}$$

因此

$$\text{sign}(s_j)=\text{sign}(\mathbf{r}_j^T\mathbf{x}) \tag{18.108}$$

这恰好就是随机超平面 LSH 的定义。

从上述推导可以看出，谷歌 SimHash 并没有改变随机投影 SimHash 的数学本质，而是利用普通哈希函数生成随机向量，从而避免显式存储高维随机投影矩阵，实现了随机超平面的隐式构造，大幅降低了计算复杂度与存储开销，更适合工业级海量文本检索。

18.3.2.0.4 谷歌 SimHash 查询 除了指纹计算之外，Google 论文更重要的贡献在于提出了一种能够支持海量数据快速检索的近重复搜索算法。论文考虑的问题如下：

笔记 对于数十亿个 64 位 SimHash 指纹，当输入一个新的指纹时，如何快速找到所有汉明距离不超过 k（通常 $k\le 3$）的历史文档？

Google 为了能够在数十亿网页中快速检索近重复文档，并没有直接对全部 64 位指纹进行暴力比较，而是提出了经典的**分块（Partition）+ 倒排索引（Inverted Index）**检索方案，其整体流程如图18.11所示。首先，将每个 64 位 SimHash 指纹均匀划分为 m 个长度相同的子块。例如，当允许最大的汉明距离阈值为 $k=3$ 时，可将 64 位指纹划分为 $m=k+1=4$ 个 16 bit 子块，即

$$\underbrace{B_1}_{16bit}\ \underbrace{B_2}_{16bit}\ \underbrace{B_3}_{16bit}\ \underbrace{B_4}_{16bit} \tag{18.109}$$

随后，对于数据库中的每一个文档，分别以每个分块作为 Key 建立倒排索引。例如某篇网页对应的四个分块分别为

$$(B_1,B_2,B_3,B_4) \tag{18.110}$$

则分别插入四张倒排表中：

$$B_1\to\{\text{Doc}_i,\cdots\} \tag{18.111}$$

$$B_2\to\{\text{Doc}_i,\cdots\} \tag{18.112}$$

依此类推，每个文档最终都会出现在 m 张倒排索引中。

当新的网页到来时，首先计算其 64 位 SimHash 指纹，并同样划分成 m 个子块，然后分别查询对应倒排索

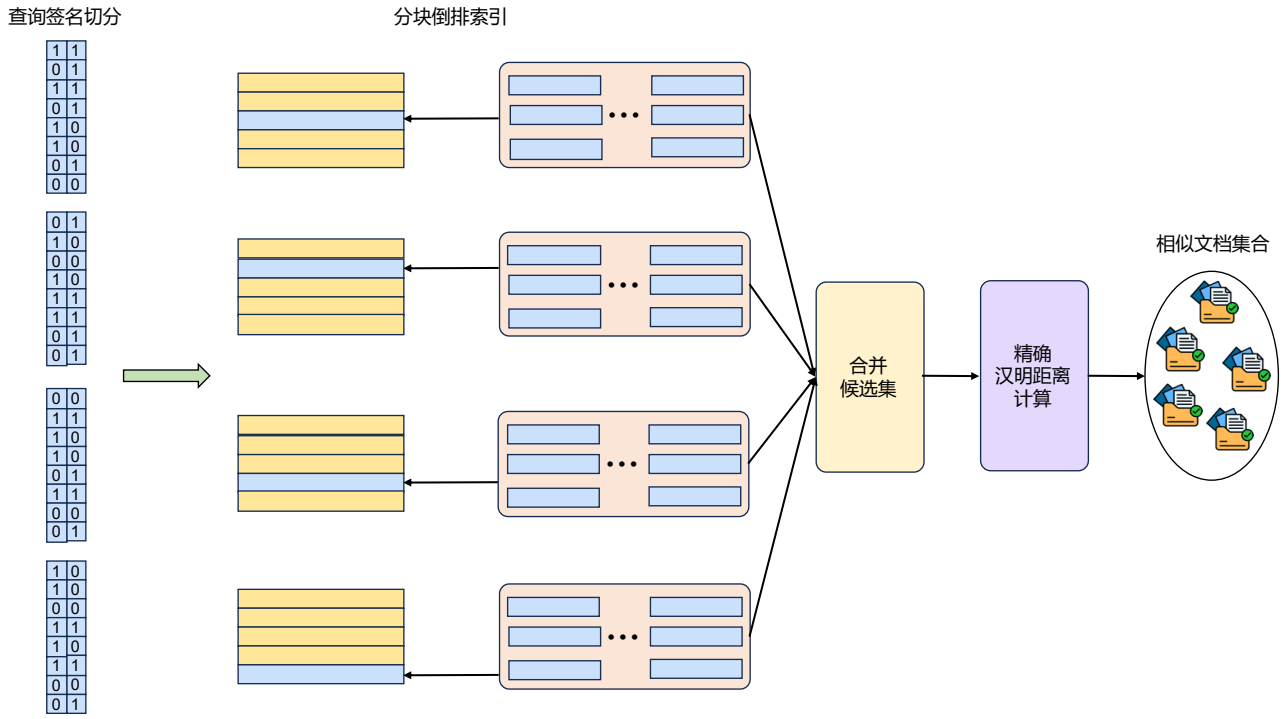

图 18.11: SimHash 分块倒排索引检索流程。

引。例如查询指纹为 (Q_1, Q_2, Q_3, Q_4)，则分别访问 Q_1, Q_2, Q_3, Q_4 对应的四条倒排链，并将得到的所有文档取并集作为候选集合。这里之所以能够保证不会漏召回满足条件的文档，关键在于**鸽笼原理（Pigeonhole Principle）**。若两条 64 位指纹之间最多只有 $k = 3$ 个 bit 不同，而整个指纹被划分成 $k + 1 = 4$ 个子块，则最多只能有 3 个分块出现错误，因此至少存在一个分块完全一致。否则若四个分块全部发生变化，则每个分块至少贡献一个 bit 错误，总错误数至少为 4，与汉明距离不超过 3 相矛盾。

因此，所有满足 $H(\mathbf{x}, \mathbf{y}) \le 3$ 的文档，一定能够在四条倒排链中的至少一条被召回，而无需扫描整个数据库。获得候选集合之后，再进一步计算候选文档与查询文档之间的真实汉明距离：

$$H(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{64} \mathbf{1}(x_i \neq y_i) \tag{18.113}$$

最后仅保留满足 $H(\mathbf{x}, \mathbf{y}) \le 3$ 的文档作为最终近重复网页。

需要注意的是，这里的倒排链长度通常远远小于整个数据库规模。例如数据库中共有 10 亿篇网页，若 16 bit 分块近似均匀分布，则一个分块仅对应约 $\frac{10^9}{2^{16}} \approx 1.5 \times 10^4$ 篇网页。查询时只需访问 4 条倒排链即可获得候选集合，其规模通常只有几万甚至几千个文档，再计算这些候选的 64 位汉明距离即可完成检索，相比直接扫描全部 10 亿个指纹，计算复杂度降低了数万倍以上。

谷歌 SimHash 因此不仅将原始随机超平面 LSH 的高维随机投影计算转化为了适合工业部署的 64 位二进制指纹生成过程，同时结合分块倒排索引，实现了海量网页近重复检测的高效检索。这一方案后来成为搜索引擎、网页爬虫、文本查重、网页去重以及大规模文档管理系统中的经典工程实现。

18.3.2.0.5 SimHash 代码示例与 Faiss 实现

下面给出一个简化版的 SimHash 实现，其核心思想与 Google 提出的方法保持一致：首先为每个特征生成固定的随机 ±1 向量，然后按照特征权重进行累加，最后依据累加结果的符号得到最终的二值指纹。

代码 18.6: SimHash 示例代码。

```
import random
import numpy as np
```

```
class SimHash:
    def __init__(self, bit_len=64, seed=42):
        self.bit_len = bit_len
        random.seed(seed)
        self.word_rand = dict() # word -> L维 [-1,1] 向量

    def _get_word_r(self, word):
        if word not in self.word_rand:
            # 生成L个±1随机数
            vec = np.random.choice([-1, 1], size=self.bit_len)
            self.word_rand[word] = vec
        return self.word_rand[word]

    def get_fingerprint(self, word_weight: dict):
        # word_weight: {单词: TF-IDF权重}
        acc = np.zeros(self.bit_len)
        for word, w in word_weight.items():
            r_vec = self._get_word_r(word)
            acc += w * r_vec
        # 符号二值化
        fp = (acc >= 0).astype(np.uint8)
        return fp

# 汉明距离
def hamming_dist(fp1, fp2):
    return np.sum(fp1 ^ fp2)
```

该实现仅用于说明 SimHash 的基本原理。在实际工业系统中，通常不会显式维护每个特征对应的随机向量，而是利用普通哈希函数直接生成对应的二进制编码，从而进一步降低内存占用并提升计算效率。目前，Faiss 提供了两种与 LSH 相关的索引结构：**IndexLSH** 和 **IndexBinaryMultiHash**。虽然二者名称都包含 LSH，但其设计目标和检索方式存在明显差异。

Faiss 中的 IndexLSH 本质上对应于随机超平面 LSH（Random Hyperplane LSH）。索引建立时，首先随机生成 k 个固定的随机超平面

$$\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_k \tag{18.114}$$

所有向量均共享这一组随机超平面。对于任意输入向量 $\mathbf{x}$，分别计算

$$h_i(\mathbf{x}) = \text{sign}(\mathbf{w}_i^\top \mathbf{x}) \tag{18.115}$$

最终得到长度为 k bit 的二进制哈希码

$$\mathbf{b} = (h_1, h_2, \cdots, h_k) \tag{18.116}$$

需要注意的是，IndexLSH 中的随机超平面在整个数据集上保持固定，并不会像随机投影树那样随着树结构递归生成新的局部分割超平面，因此整个索引不存在层次结构，仅仅完成一次全局随机映射。

查询时，同样利用这组固定随机超平面对查询向量生成二进制编码，然后通过精确的 KNN 计算其与数据库所有哈希码之间的汉明距离，再选取距离较小的候选进行精确距离计算。因此，IndexLSH 虽然采用了 LSH 思想生成二值编码，但其默认实现仍需要遍历全部数据进行汉明距离比较，本质上属于一种**压缩编码（Binary Encoding）**方法，而不是严格意义上的子线性 LSH 检索结构。

IndexLSH 最大的优点在于内存占用极低，仅需存储每个向量对应的二进制编码，因此适用于内存受限或者

需要快速生成紧凑向量表示的场景。但由于查询阶段仍需扫描全部编码，因此通常不会单独作为 ANN 索引使用，而更多作为其它索引结构的预过滤模块，进行初步的筛选过滤。

而真正实现经典 LSH 思想的是 Faiss 中的 IndexBinaryMultiHash。该方法建立多张独立哈希表，每张哈希表对应一组独立随机哈希函数。数据插入时，根据不同哈希函数分别计算哈希码，并插入对应哈希桶。查询时，仅访问查询哈希码对应的若干哈希桶，再对桶内候选计算真实汉明距离，因此能够避免扫描全部数据库，实现真正意义上的子线性近似最近邻搜索。这与 Google SimHash 论文中采用的" 分块 + 倒排索引" 思想本质一致，都属于典型的多哈希表 LSH 实现。

当然，多哈希表也带来了额外的存储开销。哈希表数量越多，近邻落入同一桶中的概率越高，Recall 通常越高；与此同时，索引大小以及构建时间也会随之增加。因此，实际工程中通常需要根据召回率和存储成本共同选择哈希表数量。

18.3.2.0.6　IndexLSH 与随机投影树比较　虽然随机投影树（RP-Tree）与随机超平面 LSH 都建立在 Johnson–Lindenstrauss 引理和随机投影思想基础之上，但两者的数据组织方式存在本质区别。随机投影树采用递归划分策略，每到一个树节点都会重新生成新的随机超平面，仅作用于当前局部数据，不断缩小数据单元的空间直径，因此其理论分析主要依赖于数据的内在维度（Intrinsic Dimension），对于低维流形数据具有较好的查询性能。

相比之下，IndexLSH 采用固定随机超平面对整个数据集一次性完成编码，不会根据局部数据分布动态调整划分方式，因此无法利用局部流形结构的信息，其理论分析更多依赖于原始空间维度。对于具有明显聚类结构的数据，固定哈希桶往往容易产生大量候选，从而影响最终召回率。二者主要区别如表18.1所示。

表 18.1: 随机投影树与 Faiss IndexLSH 的比较

对比维度	随机投影树（RP-Tree）	Faiss IndexLSH
数据结构	递归二叉树	二值编码
随机超平面	每个节点重新生成	全局固定一组
随机投影用途	递归空间划分	二进制编码生成
索引结构	分层树结构	无层次结构
查询方式	自顶向下遍历树	全量汉明距离比较
理论分析	内在维度	原始维度
局部自适应能力	强	弱
内存占用	中等	极低
适用场景	高维流形数据 ANN	二值编码、预过滤、压缩存储

综合来看，RP-Tree 更加关注随机划分带来的空间收缩性质，而 IndexLSH 更加关注利用随机投影构造紧凑的二值表示。前者属于典型的树结构 ANN 算法，后者更接近随机投影编码方法。虽然二者都利用了随机超平面，但由于索引组织方式完全不同，因此在理论分析、查询复杂度以及适用场景上均存在较大差异。

18.3.3 LSH Banding

前面介绍的 MinHash、SimHash 等方法，本质上都是为每个对象生成一个固定长度的哈希签名（Signature）。例如，MinHash 得到的是由多个 MinHash 值组成的签名向量，SimHash 得到的是长度通常为 64 bit 或 128 bit 的二进制指纹。然而，仅仅拥有哈希签名并不足以完成近似最近邻搜索，因为如果直接要求两条签名完全一致，则召回率往往极低；而如果逐位比较所有签名，又需要遍历整个数据库，无法达到加速检索的目的。

因此，LSH 真正用于 ANN 检索时，还需要引入一种称为 **Banding（分段）** 的哈希策略。Banding 的核心思想如下：

定义 18.8

与普通哈希不同的是，LSH 不是要求整个签名完全相同，而是只要某一个连续分段完全一致，就认为两个对象可能相似，并将其映射到同一个哈希桶中。

我们假设每个对象最终得到长度为 $n = br$ 的哈希签名，其中 b 是 Band（分段）的数量，r：每个 Band 中包含的哈希位数（Rows）。于是整个签名可以写成

$$\underbrace{\cdots}_{r}\ \underbrace{\cdots}_{r}\ \cdots\ \underbrace{\cdots}_{r} \tag{18.117}$$

共划分为 b 个 Band。随后，对于每一个 Band，分别采用普通哈希函数计算哈希值，并建立对应的哈希桶。如图18.12所示，同一个 Band 中的 r 个哈希值共同组成一个 Key，若两个对象在某个 Band 上的哈希值完全一致，则会被映射到同一个哈希桶中。比如图18.12中两个相同的 key 被映射到了第 2 个桶中，称为相似对象，而两个不同的 key 因为误差的原因被映射到了第 5 个分桶中，从而构成了 false positive 对象。

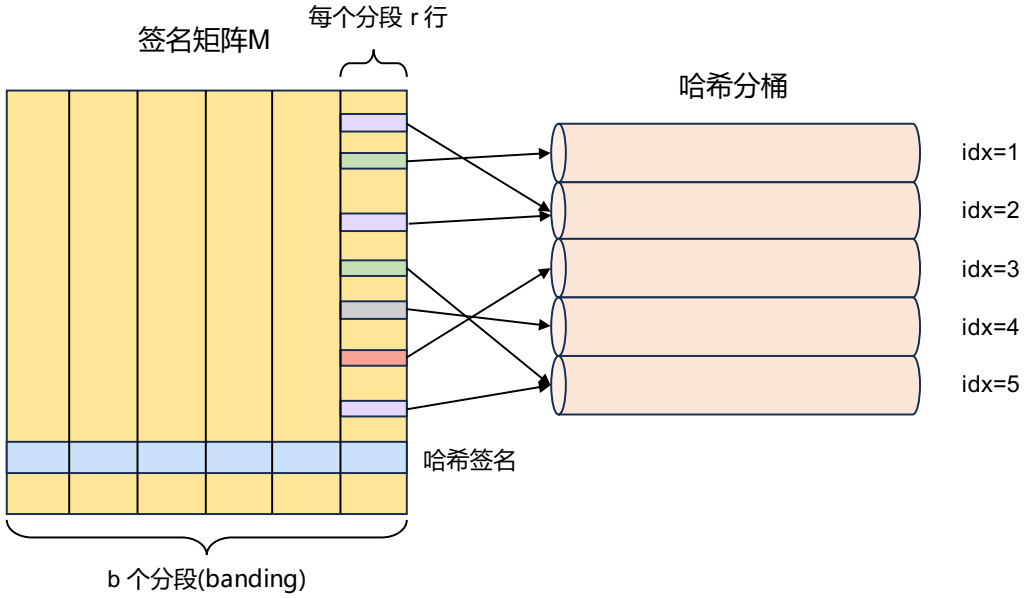

图 18.12: LSH Banding 过程示意图。

设两个对象之间每一位哈希值相同的概率为

$$t = P(h(x) = h(y)) \tag{18.118}$$

对于 MinHash 而言，$t = J(A, B)$ 即 Jaccard 相似度；对于 SimHash 而言，$t = 1 - \frac{\theta}{\pi}$ 对应余弦相似度。下面推导两个对象最终进入同一个哈希桶的概率。

首先考虑某一个 Band。由于 Band 内包含 r 个独立哈希函数，因此两个对象只有当这一 Band 内所有 r 位哈希值全部一致时，才会映射到同一个 Bucket。因此：

$$P(\text{Band 匹配}) = t^r \tag{18.119}$$

相反，一个 Band 不匹配的概率为

$$P(\text{Band 不匹配}) = 1 - t^r \tag{18.120}$$

整个 LSH 一共有 b 个 Band，只有当所有 Band 全部不匹配时，两个对象才不会进入任何共同的哈希桶，因此

$$P(\text{所有 Band 均不匹配}) = (1 - t^r)^b \tag{18.121}$$

于是我们能得到如下的概率公式：

定义 18.9 (LSH 碰撞概率公式)

至少存在一个 Band 匹配（即能够召回）的概率为

$$P(\text{Candidate}) = 1 - (1 - t^r)^b \tag{18.122}$$

这便是 LSH Banding 最经典的概率公式。

从上述公式可以进一步分析 Banding 参数对召回概率的影响。当两个对象非常相似（$t \to 1$）时，即：

$$1 - (1 - t^r)^b \to 1, \tag{18.123}$$

说明高相似对象几乎一定能够进入同一个哈希桶。而当两个对象相似度较低（$t \to 0$）时，

$$1 - (1 - t^r)^b \to 0, \tag{18.124}$$

说明低相似对象极少会产生碰撞。在中间区域中，该函数会呈现典型的"S 型阶跃曲线"，即相似度略高于某个阈值时，碰撞概率迅速由接近 0 上升至接近 1。因此，LSH Banding 实际上相当于为相似度构造了一个近似的二值判别器。相比直接比较全部签名，Banding 能够在保持较高召回率的同时，大幅减少哈希桶中的候选数量。

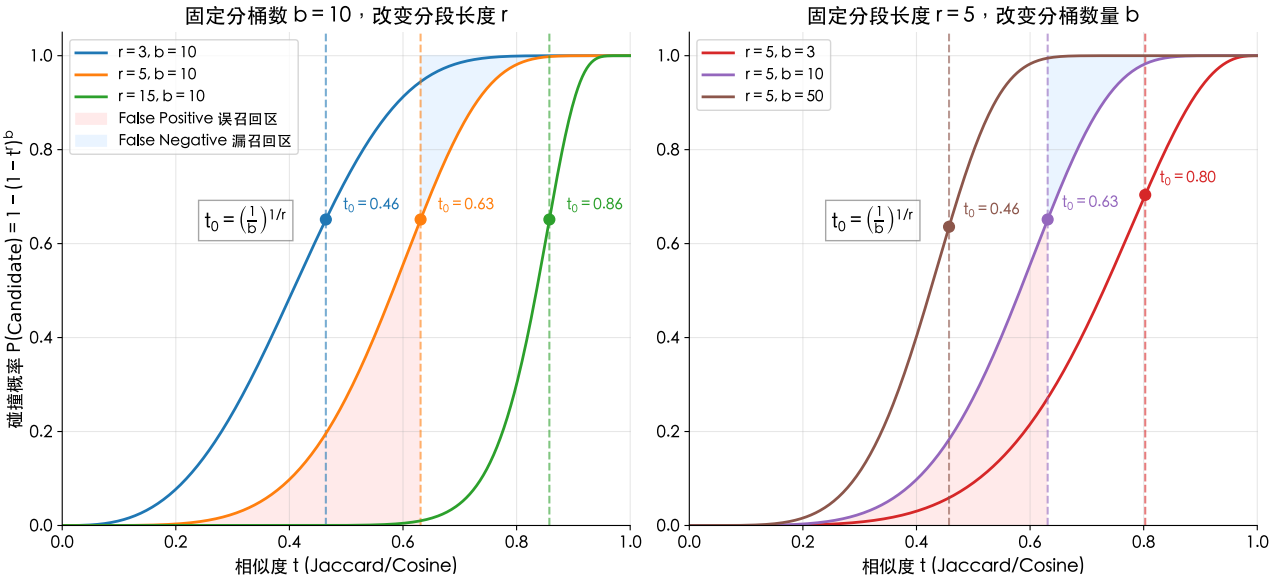

图 18.13: LSH Banding 碰撞概率曲线与超参数 r 和 b 的关系。

为了进一步理解超参数 r 与 b 对 LSH 性能的影响，可以固定其中一个参数，分析另一个参数对碰撞概率曲线的影响。首先考虑分段长度 r。从图18.13左图可以看到，当 r 由 3 逐渐增大到 5，再增大到 15 时，整条 S 型曲线不断向右移动。其原因可以直接由前文碰撞概率公式得到解释。当 r 增大时，一个分段内要求更多哈希 bit 位同时一致，因此单个分段发生碰撞的概率 t^r 会迅速下降，只有相似度更高的对象才能保持较大的碰撞概率。因此，对于固定的真实相似度阈值 t_0 而言，LSH 对候选对象的筛选条件变得更加严格。例如，在图18.13左图中，以 $t_0 = 0.63$ 作为真实近邻判定阈值。当 r 增大以后，低于该阈值的对象碰撞概率明显下降，因此误召回（False Positive）减少；与此同时，一部分原本属于真实近邻、但相似度仅略高于 t_0 的对象，其碰撞概率也会有所下降，因此漏召回（False Negative）相应增加。换句话说，较大的 r 能够获得更高的 Precision，而较小的 r 通常能够获得更高的 Recall。

另一方面，分段数量 b 决定了共有多少次发生碰撞的机会。从图18.13右图可以看到，当 b 由 3 增加到 10，再增加到 50 时，S 型曲线整体不断向左移动。由碰撞概率公式可知，随着 b 不断增大，碰撞概率整体单调增大，即对象只要在任意一个分段发生碰撞即可进入候选集合，因此更多对象能够成功进入候选集，Recall 得到提高；与此同时，候选集合规模也会随之增大，从而带来更多精确距离计算，查询延迟与计算开销也会相应增加。

综合来看，参数 r 主要控制每一个 Band 内部匹配条件的严格程度，而 b 则决定系统提供多少次独立的碰撞机会。较大的 r 对应更严格的判定条件，能够提高 Precision，但 Recall 有所下降；较大的 b 则能够提高 Recall，

但同时增加候选集合规模以及后续精排计算量。因此，实际工程中通常需要联合调节 r 与 b，在 Recall、Precision 以及查询效率之间取得平衡。

进一步分析碰撞概率函数，还可以得到 LSH Banding 曲线的近似拐点位置。设希望碰撞概率达到约一半，即

$$P_{\text{collide}}(t_0) \approx \frac{1}{2} \tag{18.125}$$

则有

$$1-(1-t_0^r)^b = \frac{1}{2} \tag{18.126}$$

整理得到

$$(1-t_0^r)^b = \frac{1}{2} \tag{18.127}$$

进一步化简为

$$t_0^r = 1 - 2^{-1/b}. \tag{18.128}$$

当 b 通常取十几甚至几十时，可以利用一阶 Taylor 展开

$$2^{-1/b} = e^{-\frac{\ln 2}{b}} \approx 1 - \frac{\ln 2}{b} \tag{18.129}$$

于是有

$$t_0^r \approx \frac{\ln 2}{b} \tag{18.130}$$

由于 $\ln 2 \approx 0.693$ 接近于常数 1，因此工程实践中通常进一步近似写成

$$t_0 \approx \left(\frac{1}{b}\right)^{1/r} \tag{18.131}$$

该近似公式虽然忽略了常数 $\ln 2$，但能够较准确地描述 S 型曲线由低碰撞概率快速跃迁到高碰撞概率的位置，因此被广泛用于 LSH 参数的初步设计。从公式可以看出，增大 r 会提高碰撞阈值，使曲线向右移动；增大 b 则降低碰撞阈值，使曲线向左移动，这与图18.13中的实验现象完全一致。

总的来说，MinHash、SimHash 等算法负责构造具有局部敏感性质的哈希签名，而 LSH Banding 进一步将这些签名组织为多个分段并建立哈希桶，实现由“相似度估计”向“近似最近邻搜索”的转换。前者决定了签名本身是否能够保持局部相似性，后者则决定了检索阶段候选集合的规模以及召回性能。因此，在几乎所有工业级 LSH 系统中，Banding 都是不可或缺的重要组成部分，也是 LSH 能够实现子线性近似最近邻搜索的关键所在。

18.4 乘积量化 (PQ)

乘积量化（Product Quantization，PQ）最早由 Jégou 等人在 2011 年发表于《Product Quantization for Nearest Neighbor Search》[4] 中提出，是目前工业界近似最近邻（Approximate Nearest Neighbor，ANN）搜索中应用最广泛的向量压缩算法之一，也是 Faiss、Milvus、ScaNN 等主流 ANN 库的重要基础。

PQ 提出的背景来源于一个非常现实的问题：随着 Embedding 模型的发展，向量维度不断提高，一个推荐系统往往需要存储数亿甚至数十亿个高维 Embedding。例如，一个 128 维 Float32 向量需要 $128 \times 4 = 512$ Byte。若数据库包含 10^9 个向量，仅存储 Embedding 便需要约 $512 \times 10^9 \approx 512$ GB。如此大的存储规模不仅带来了巨大的内存压力，同时暴力计算查询向量与所有数据库向量之间的欧氏距离，其计算复杂度也达到 $\mathcal{O}(ND)$，其中 N 表示数据库向量数量，D 表示向量维度。在工业界推荐系统中，如果按照上述传统的方式，无论是 Embedding 的存储还是检索查询，都会面临极高的存储和耗时压力。因此，如何在尽可能保持向量距离关系的前提下，对高维 Embedding 进行压缩，并实现快速距离计算，成为 ANN 研究的重要问题。

最直接的压缩方法是使用 Vector Quantization（VQ）进行向量量化，关于 VQ 相关的内容我们也在本书前面的第9章生成式推荐模块中进行过介绍。这里我们简要回顾一下 VQ 的方法。假设整个向量空间训练得到一个包含 K 个聚类中心的码本（Codebook）：

$$\mathcal{C} = \{\mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_K\}, \tag{18.132}$$

则任意向量 $\mathbf{x}$ 都可以表示为距离最近的聚类中心

$$q(\mathbf{x}) = \arg\min_{\mathbf{c}_i \in \mathcal{C}} \|\mathbf{x} - \mathbf{c}_i\| \tag{18.133}$$

这样，每个向量仅需要保存对应聚类中心的编号即可，大大降低了存储开销。然而，当向量维度较高时，普通的 VQ 方法存在明显缺陷。为了获得较小的量化误差，通常需要训练大量的聚类中心，而聚类中心数量也会随着向量维度增加呈指数增长。例如，当每个维度希望具有 16 种量化状态时，一个 128 维空间理论上需要 $16^{128} = 2^{28}$ 个聚类中心，这在存储与训练上都是完全不可接受的，因此普通 Vector Quantization 难以直接应用于高维向量。

基于普通聚类存在的上述问题，PQ 算法进行了如下的改进：

定义 18.10

虽然整个高维空间难以直接量化，但若将高维空间划分为多个低维子空间，则每个子空间均可以独立完成向量量化。PQ 算法正是基于这一想法展开的。PQ 算法的优化目标为：

$$\min_{\mathcal{C}_1,\cdots,\mathcal{C}_m} \sum_{i=1}^{N}\sum_{j=1}^{m} \|\mathbf{u}_j(\mathbf{x}_i) - \mathbf{c}_{j,a_{ij}}\|^2 \tag{18.134}$$

其中 N 是向量的总个数，$\mathcal{C}_j$ 是第 j 个子空间的码本，$\mathbf{u}_j(\mathbf{x}_i) \in \mathbb{R}^{D/m}$ 是子空间中的向量，a_{ij} 满足：

$$a_{ij} = \arg\min_{k} \|\mathbf{u}_j(\mathbf{x}_i) - \mathbf{c}_{j,k}\|^2 \tag{18.135}$$

♣

假设原始向量维度为 D，PQ 算法首先将其均匀划分为 m 个互不重叠的子空间，每个子空间维度为 $D^* = \frac{D}{m}$。因此，一个向量可以写成

$$\mathbf{x} = [\mathbf{u}_1(\mathbf{x}), \mathbf{u}_2(\mathbf{x}), \cdots, \mathbf{u}_m(\mathbf{x})] \tag{18.136}$$

其中 $\mathbf{u}_j(\mathbf{x}) \in \mathbb{R}^{D^*}$ 表示第 j 个子向量。例如，一个 128 维 Embedding，当设置 $m = 8$ 时，每个子空间仅包含 16 维的向量。

对于每一个子空间，我们可以分别独立训练一个该子空间对应的码本。设第 j 个子空间对应的码本表示为

$$\mathcal{C}_j = \{\mathbf{c}_{j,1}, \mathbf{c}_{j,2}, \cdots, \mathbf{c}_{j,K}\} \tag{18.137}$$

其中每个聚类中心均由 K-Means 训练得到。因此，第 j 个子向量的量化结果为

$$q_j(\mathbf{x}) = \arg\min_{\mathbf{c} \in \mathcal{C}_j} \|\mathbf{u}_j(\mathbf{x}) - \mathbf{c}\| \tag{18.138}$$

然后整个 PQ 编码就可以表示为

$$q(\mathbf{x}) = [q_1(\mathbf{x}), q_2(\mathbf{x}), \cdots, q_m(\mathbf{x})] \tag{18.139}$$

也就是说，一个数据库向量最终只需要保存 $[i_1, i_2, \cdots, i_m]$ 共 m 个整数索引就可以了，而无需保存完整浮点向量。这样就能显著节省原始 Embedding 的存储空间。例如，工业界常见的配置可以是 $m = 8, K = 256$。这里由于每个子码本共有 256 个聚类中心，每个子空间就只需要 8 bit 来表示聚类中心编号。因此，整个 128 维 Float32 向量最终只需要 $8 \times 8 = 64$ bit $= 8$ Byte 即可完成编码。相比原始 128 维 Float32 向量的 512 Byte，压缩率达到 $\frac{512}{8} = 64$。也就是说，在保持较低量化误差的情况下，PQ 能够实现约 64 倍的存储压缩，这也是其能够广泛应用于十亿级向量检索的重要原因。

进一步来看，虽然每个子空间仅包含 K 个聚类中心，但整个向量空间对应的组合码字数量实际上可达到 K^m。例如，当 $m = 8$，$K = 256$ 时，$256^8 = 2^{64}$，意味着整个 PQ 系统理论上能够表示超过 10^{19} 种不同的重建向量，而实际仅需存储 $m \times K = 8 \times 256 = 2048$ 个聚类中心即可。换句话说，PQ 并不显式存储完整的 K^m 个码字，而是利用多个低维子码本的笛卡尔积（Cartesian Product）隐式构造出一个规模指数增长的码本（Implicit Codebook）。因此，PQ 能够以线性增长的存储开销获得指数级增长的表示能力，这也是“Product Quantization（乘积量化）”这一名称的由来。

此外，需要注意的是，PQ 与前面介绍的 MinHash、SimHash 等局部敏感哈希（LSH）方法属于两类不同的近似最近邻搜索技术。MinHash 主要面向集合数据，利用 Jaccard 相似度进行近似检索；SimHash 则主要针对余弦相似度进行局部敏感哈希。而 PQ 属于向量量化（Vector Quantization）方法，主要用于**欧氏距离（L2 Distance）**

或内积距离（Inner Product）下的近似最近邻搜索，其核心思想并非通过哈希碰撞筛选候选，而是利用向量量化对数据库向量进行高效压缩，并借助查找表快速近似距离计算。

18.4.1 距离计算：SDC 与 ADC

完成 PQ 编码之后，我们可能会产生一个新的问题：

笔记 数据库中的向量已经通过 PQ 算法压缩为离散的码字索引，然后呢？在查询阶段应该如何计算查询向量与数据库向量之间的距离？

一种最直接的思路是首先利用 PQ 码字重建出向量，然后计算两者之间的欧氏距离。然而，这样需要恢复完整浮点向量，计算开销较大，也失去了 PQ 快速检索的优势。因此，PQ 论文提出了两种距离计算方式：**对称距离计算**（Symmetric Distance Computation，SDC）与**非对称距离计算**（Asymmetric Distance Computation，ADC）。

18.4.1.0.1 SDC SDC 的思想十分直接：查询向量 x 和数据库向量 y 均首先进行 PQ 编码，再利用对应码字之间的距离近似原始欧氏距离。设

$$q(x) = [q_1(x), q_2(x), \cdots, q_m(x)] \tag{18.140}$$

$$q(y) = [q_1(y), q_2(y), \cdots, q_m(y)] \tag{18.141}$$

则两者之间的距离可近似表示为

$$\tilde{d}(x, y) = d(q(x), q(y)) = \sqrt{\sum_{j=1}^{m} d(q_j(x), q_j(y))^2} \tag{18.142}$$

由于每个子空间只有 K 个聚类中心，因此对于第 j 个子空间，可以通过预处理的方式计算出所有聚类中心之间的距离 $d(c_{j,i}, c_{j,k})$，从而形成一个 $K \times K$ 大小的查找表（Lookup Table）。因此，查询过程中无需计算任何浮点距离，仅需要读取对应码字之间的距离并累加即可。比如当 $K = 256$ 时，每个子空间对应的查找表仅包含 $\frac{1}{2} \times 256 \times 255 = 32640$ 个距离值，可全部提前存储于内存中，因此查询速度非常快。

然而，SDC 存在一个明显的问题：查询向量同样经过量化，因此查询端也会引入量化误差。设

$$x = q(x) + e_x, \tag{18.143}$$

$$y = q(y) + e_y, \tag{18.144}$$

则最终计算的实际上是 $\|q(x) - q(y)\|$，而不是原始距离 $\|x - y\|$。因此，数据库和查询向量都会产生量化误差。

18.4.1.0.2 ADC PQ 论文提出的核心创新实际上并不是 PQ 编码本身，而是非对称距离计算（Asymmetric Distance Computation，ADC）。ADC 认为数据库规模通常达到数亿甚至数十亿，因此必须压缩；而查询向量一次通常只有一个，没有必要进行量化。因此，仅数据库向量进行 PQ 编码，而查询向量始终保持原始浮点表示。

设数据库向量编码结果为

$$q(y) = [c_{1,i_1}, c_{2,i_2}, \cdots, c_{m,i_m}] \tag{18.145}$$

查询向量保持原始形式

$$x = [u_1(x), u_2(x), \cdots, u_m(x)] \tag{18.146}$$

则两者距离近似为

$$\tilde{d}(x, y) = \|x - q(y)\| = \sqrt{\sum_{j=1}^{m} \|u_j(x) - c_{j,i_j}\|^2} \tag{18.147}$$

ADC 与 SDC 的计算差异如图18.14所示。可以看到，相比起 SDC，ADC 的距离计算中仅在数据库一侧进行了量化，因此 ADC 只引入了数据库端的量化误差。如果给数据库端向量引入误差形式表示为 $y = q(y) + e$，则 ADC 的距离 $\|x - q(y)\|$ 比起 SDC 少了查询向量部分的量化误差，因此 ADC 这种距离估计更加准确。论文中的实验

也证明了在相同编码长度下，ADC 的 Recall 通常明显优于 SDC，因此在工业界几乎所有 PQ 系统均采用 ADC 作为默认距离计算方式。

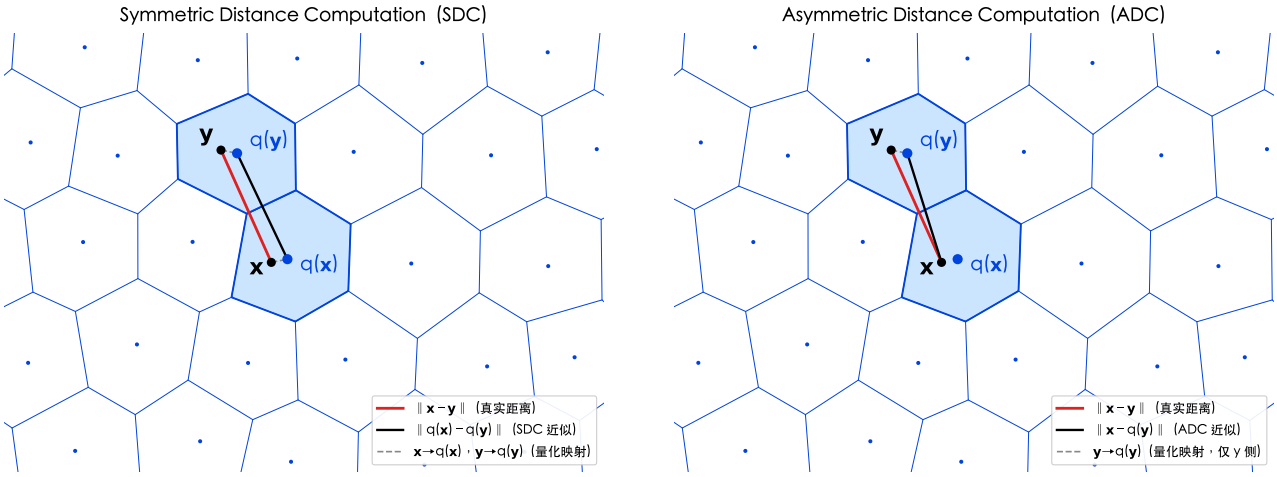

图 18.14: ADC 与 SDC 的距离计算差异对比。

除此之外，ADC 优势还在于虽然查询向量没有编码，但仍然可以做到几乎无需浮点距离计算。对于固定查询向量 x，首先分别计算每个子空间到所有聚类中心之间的距离：

$$L_j(i) = \|u_j(x) - c_{j,i}\|^2 \tag{18.148}$$

其中 $j = 1, \cdots, m$，$i = 1, \cdots, K$。于是，对于每个查询，仅需构造 m 个查找表。整个查找表可以表示为矩阵形式：

$$\mathbf{L} = \begin{bmatrix} L_1(1) & \cdots & L_1(K) \\ L_2(1) & \cdots & L_2(K) \\ \vdots & & \vdots \\ L_m(1) & \cdots & L_m(K) \end{bmatrix} \tag{18.149}$$

例如，当 $m = 8, K = 256$ 时，整个查找表的大小仅为 $8 \times 256 = 2048$ 个浮点数。随后，对于任意数据库向量，由于 PQ 编码仅保存 $[i_1, i_2, \cdots, i_m]$ 位置索引，所以整个 ADC 距离计算就会退化为如下的形式：

$$\tilde{d}(x, y) = \sqrt{L_1(i_1) + L_2(i_2) + \cdots + L_m(i_m)} \tag{18.150}$$

在整个过程中无需重新计算欧氏距离，仅需 m 次数组查表与累加即可。

接下来我们对 ADC 的计算复杂度进行分析。假设数据库包含 N 个向量，向量维度为 D。如果采用暴力搜索，则一次查询需要与数据库中的全部 N 个向量计算完整的欧氏距离，其时间复杂度为 $\mathcal{O}(ND)$。采用 ADC 之后，首先需要为查询向量建立距离查找表，其时间复杂度为 $\mathcal{O}(mKD^*)$，其中 m 表示子空间个数，K 表示每个子空间中的聚类中心数量，$D^* = \frac{D}{m}$ 表示每个子空间的维度。由于 $mD^* = D$，因此建立查找表的复杂度可进一步化简为 $\mathcal{O}(KD)$。随后，查询向量仍需与数据库中的每个向量计算一次距离，但每次距离计算仅需进行 m 次查表和累加操作，因此数据库扫描阶段的时间复杂度为 $\mathcal{O}(mN)$。综合来看，ADC 一次查询的总时间复杂度为 $\mathcal{O}(KD + mN)$。其中，KD 通常仅为几千，而 m 一般取 8 或 16，因此距离计算由原来的 D 维浮点运算降低为 m 次整数查表。例如，对于 128 维的 Embedding，取 $D = 128$，$m = 8$ 时，每个数据库向量仅需 8 次查表即可完成距离计算，相较于直接计算 128 维欧氏距离能够获得数量级上的速度提升。

18.4.2 IVFADC 框架

虽然 ADC 能够显著降低单个向量的距离计算开销，但其仍然存在一个根本问题：对于每一次查询，仍然需要扫描数据库中的全部向量，因此 PQ 算法总体的时间复杂度仍为 $\mathcal{O}(N)$，其中 N 表示数据库中的向量数量。而当数据库规模达到数亿甚至数十亿时，即使每个向量只需要进行 m 次查表运算，全向量数据库扫描仍然无法

满足毫秒级在线检索的需求。因此，仅依靠 PQ 本身只能解决“距离计算快”的问题，而无法解决“候选数量过多”的问题。为了进一步降低搜索复杂度，PQ 算法的论文进一步提出了**倒排文件结合乘积量化**（Inverted File with Asymmetric Distance Computation，IVFADC）结构，其基本思想如下：

定义 18.11

使用倒排文件可以先缩小整体的搜索范围，然后再利用 PQ 就能完成距离的快速计算。

IVFADC 的整体架构如图18.15所示，整体包括数据库离线建索引和在线查询两个环节。在建索引环节，IVF（Inverted File）首先利用一次粗粒度的 K-Means 聚类，将整个向量空间划分为若干个区域。假设训练得到的粗聚类中心为

$$\mathcal{C} = \{\mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_{n_{\text{list}}}\} \tag{18.151}$$

其中 n_{list} 表示倒排列表（Inverted List）的数量。对于数据库中的任意向量 $\mathbf{y}$，首先找到距离最近的粗聚类中心：

$$c(\mathbf{y}) = \arg\min_{\mathbf{c}_i} \|\mathbf{y} - \mathbf{c}_i\| \tag{18.152}$$

随后，将该向量加入对应聚类中心维护的倒排链表（Posting List）中。最后整个数据库被划分为 n_{list} 个不同的倒排列表。

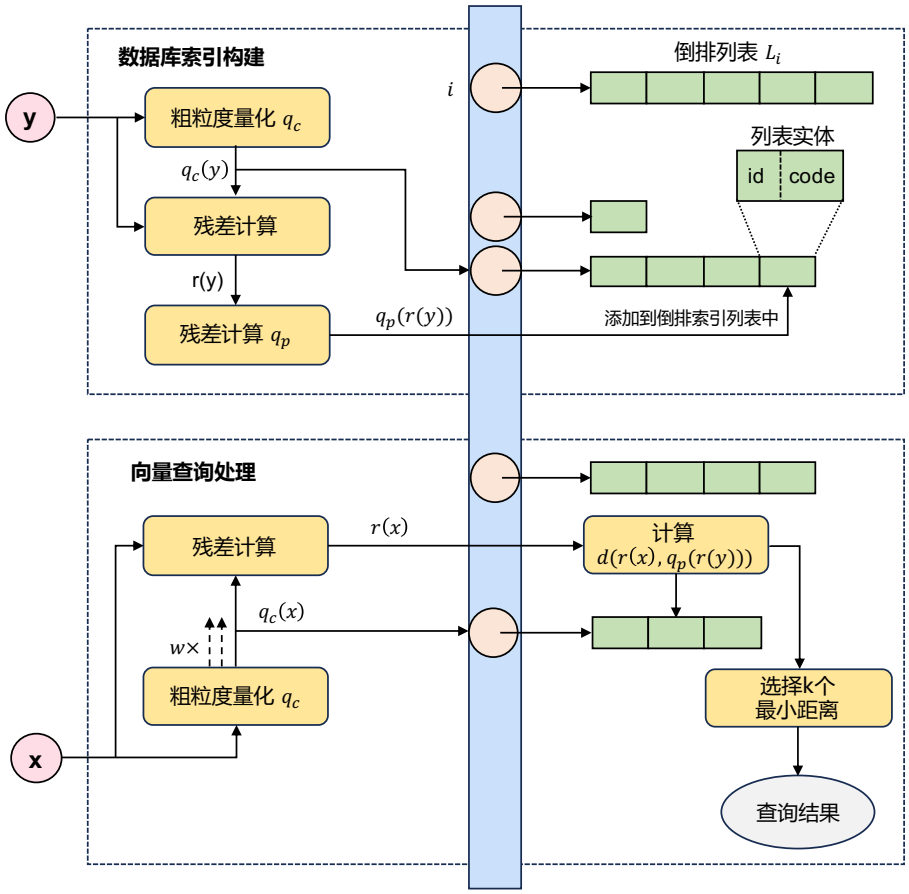

图 18.15: 倒排文件结合非对称距离计算（IVFADC）整体架构示意图。

在查询阶段结合 IVF 与 ADC 之后，一次完整的查询流程包括如下步骤：

1. 计算查询向量与所有粗聚类中心之间的距离；
2. 选择距离最近的 n_{probe} 个倒排列表；
3. 对每个倒排列表中的 PQ 编码建立查找表；
4. 利用 ADC 快速计算候选向量距离；
5. 合并所有候选结果，返回距离最近的 Top-k 向量。

具体来说，在计算查询向量与所有粗聚类中心之间的距离时，仅选择距离最近的 n_{probe} 个倒排列表继续搜索，而无需遍历全部数据库。比如，当数据库共有 10^9 个向量时，倒排列表数量为 $n_{\text{list}} = 2^{16} = 65536$，那么每个倒排列表平均包含约 $\frac{10^9}{65536} \approx 1.5 \times 10^4$ 个向量。如果查询时仅访问 $n_{\text{probe}} = 32$ 个倒排列表，则实际参与距离计算的向量数量约为 $32 \times 1.5 \times 10^4 \approx 5 \times 10^5$。这相比起扫描全部数据库，计算量降低了大约三个数量级。

除了引入 IVF 做粗粒度的量化之外，PQ 论文还引入了**残差量化（Residual Quantization）**的思想来进行一层更细粒度的残差编码。假设数据库的向量属于第 i 个倒排列表，那么它的粗聚类中心为 $\mathbf{c}_i$。首先计算残差向量 $\mathbf{r} = \mathbf{y} - \mathbf{c}_i$ 随后，对残差向量而不是原始向量进行 PQ 编码，即：

$$q(\mathbf{r}) = [q_1(\mathbf{r}), \cdots, q_m(\mathbf{r})] \tag{18.153}$$

因此，数据库中实际保存的是 (cluster id, PQ code) 这种 tuple 格式的数据而不是原始向量。

在查询时，查询向量与数据库向量之间的距离可近似表示为

$$d(x, y) = \|\mathbf{x} - \mathbf{y}\| \approx \|\mathbf{x} - (\mathbf{c}_i + q(\mathbf{y} - \mathbf{c}_i))\| = \|(\mathbf{x} - \mathbf{c}_i) - q(\mathbf{y} - \mathbf{c}_i)\| = d(\mathbf{x} - \mathbf{c}_i, q(\mathbf{y} - \mathbf{c}_i)) \tag{18.154}$$

所以只需要对查询向量 $\mathbf{x}$ 和数据库向量 $\mathbf{y}$ 统一减去全局的偏置项 $\mathbf{c}_i$，然后进行传统的距离计算即可。相比起直接量化原始向量，残差向量通常具有更小的范数。这是因为粗聚类已经消除了数据的大部分全局偏移，不同向量仅保留局部细节，因此残差空间会更加集中，也更容易利用有限数量的聚类中心进行表示。从另一个角度去看，同样大小的 PQ 码本，在残差空间中的量化误差明显更小，因此 Recall 通常也能够进一步提升。

总的来说，在 IVFADC 框架中，IVF 在某种程度上承担了“粗粒度召回”的作用，而 PQ 则负责进行后续的“精细化排序”。因此，IVFADC 框架也可以看做是一种类似推荐系统中“粗排 + 精排”的两阶段式检索框架：

$$\text{粗聚类召回} \Longrightarrow \text{PQ 快速距离计算} \Longrightarrow \text{Top-}k \text{ 排序}$$

整个 IVFADC 框架不仅降低了候选规模，还降低了单个候选的距离计算成本。

18.4.3 Faiss 中的 PQ 实现

随着 PQ 算法的发展，其已经成为现代 ANN 库中最重要的向量压缩方法之一。目前 Facebook AI Research 开源的 Faiss 几乎完整实现了 PQ 论文中的所有核心思想，并在此基础上进一步扩展出了多种适用于不同场景的索引结构。最基础的索引为 **IndexPQ**。该索引直接对全部数据库向量进行 PQ 编码，查询时利用 ADC 完成距离计算。其优点是索引结构简单、内存占用低，适用于百万级左右的数据规模。然而，由于 IndexPQ 仍然需要遍历整个数据库，因此其时间复杂度仍为 $\mathcal{O}(N)$，因此随着数据库规模不断增大，查询延迟也会线性增长。

为了进一步降低搜索复杂度，Faiss 提供了 **IndexIVFPQ**。该索引首先利用 IVF 建立倒排索引，然后仅对选中的若干倒排列表中的向量执行 ADC 距离计算，因此整体检索流程可以表示为

$$\text{粗聚类召回} \Longrightarrow \text{Residual PQ 编码} \Longrightarrow \text{ADC 距离计算} \Longrightarrow \text{Top-}k \text{ 排序}$$

这几个步骤。其中，查询阶段最重要的两个超参数分别为倒排列表数量 n_{list} 和搜索列表数量 n_{probe}。n_{list} 决定了整个向量空间被划分为多少个倒排区域。通常 n_{list} 越大，每个倒排列表中的向量数量越少，因此每次查询需要扫描的候选集合更小；但与此同时，粗聚类训练成本和索引规模也会相应增加。n_{probe} 表示查询阶段实际访问多少个倒排列表。当 n_{probe} 较小时，查询速度最快，但容易遗漏真实近邻；随着 n_{probe} 不断增加，召回率逐渐提高，但查询延迟也会同步增加。因此，在工业界通常通过调整 n_{probe} 实现 Recall 与 Latency 之间的权衡。

除此之外，Faiss 还提供了 **IndexIVFOPQ**。相比普通 IndexIVFPQ，该索引首先利用 OPQ 学习一个正交旋转矩阵，对原始 Embedding 进行旋转，再完成 Residual PQ 编码。由于旋转后不同子空间之间更加独立，因此通常能够进一步降低量化误差，在编码长度保持不变的情况下获得更高的 Recall。目前，IndexIVFOPQ 也是 Faiss 推荐使用的默认 PQ 方案之一。

对于更大规模的数据集，Faiss 还进一步支持 GPU 版本的 PQ 检索。由于 ADC 距离计算主要由查表和距离累加两个步骤组成，其计算模式具有高度规则性，非常适合 GPU 进行 SIMD（Single Instruction Multiple Data）并行计算。因此，在现代 GPU 上，可以同时完成数百万甚至数千万个 PQ 编码向量的距离计算，使 PQ 能够满足十亿级向量检索场景下毫秒级在线服务的需求。

综合来看，目前 Faiss 中常见的 PQ 索引主要包括如下几种：

- **IndexPQ**：直接利用 PQ 编码全部数据库向量，实现简单，但仍需线性扫描全部数据；
- **IndexIVFPQ**：结合 IVF 与 Residual PQ，仅搜索部分倒排列表，是目前工业界最常用的 PQ 索引之一；
- **IndexIVFOPQ**：在 IndexIVFPQ 基础上进一步加入 OPQ 旋转，能够进一步降低量化误差，提高检索 Recall；
- **GPU PQ**：利用 GPU 完成查找表的构建以及 ADC 距离计算，进一步提升大规模 ANN 检索吞吐能力。

18.5 HNSW

如果说乘积量化（Product Quantization，PQ）是从**向量压缩**的角度解决近似最近邻搜索问题，那么**分层可导航小世界图**（Hierarchical Navigable Small World，HNSW）则代表了另一条完全不同的技术路线，即利用**图结构（Graph-based Index）**实现高效向量检索。HNSW 最早由 Malkov 和 Yashunin 发表于 2018 年的 IEEE TPAMI 论文《Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs》[6]，目前已经成为 Milvus、Qdrant、Weaviate、Vespa 等主流向量数据库默认采用的 ANN 索引之一，也是工业界应用最广泛的图索引算法。

相比于 KD 树、Ball Tree、随机投影树等空间划分方法，HNSW 并不试图递归划分整个向量空间，而是直接把整个数据集组织成为一张近邻图（Nearest Neighbor Graph）。图中的每一个节点对应一个数据向量，两节点之间的边表示它们具有较高的局部相似性。查询时无需遍历整个数据库，而是在图中不断沿着距离查询向量越来越近的节点进行导航（Navigation），最终快速到达目标邻域，因此又称为 **Graph-based ANN**。

HNSW 是在可导航小世界网络（Navigable Small World，NSW）的基础上发展而来的。所谓的小世界网络（Small World Network, SWN），最早来源于社会网络研究。它主要具备以下两个重要的性质：

- 任意两个节点之间通常只需要经过很少几次跳转（Short Path）即可到达；
- 节点之间具有较高的局部聚类系数（High Clustering）。

在我们的现实生活中，社交网络便具有典型的小世界特性。例如，一个人虽然只认识有限数量的朋友，但往往能够通过几层好友关系联系到世界上的绝大多数人，即著名的”六度分隔（Six Degrees of Separation）”理论。HNSW 正是借鉴了这一思想，希望构建一张既具有良好局部连接，又具有少量远距离连接（Long-range Link）的近邻图，使得搜索过程能够快速跳转至目标区域。

然而，普通 NSW 仍然存在两个明显的问题。首先，NSW 通常采用单层图结构。当数据规模不断增大时，虽然图中存在远距离连接，但搜索仍然容易陷入局部最优（Local Minimum），导致需要访问大量节点才能找到真正的最近邻。其次，由于所有节点均处于同一层，搜索只能依靠局部贪心不断前进，缺乏一种能够快速缩小搜索范围的全局导航机制，因此随着数据规模增长，搜索效率会逐渐下降。

为了解决这一问题，HNSW 进一步借鉴了经典数据结构中的跳表（Skip List）思想，引入了**多层图结构（Hierarchical Graph）**。跳表通过多层索引实现一维有序序列的快速查找，而 HNSW 则将这一思想推广到了高维向量空间。整个图由多层近邻图组成，高层图节点较少但连接跨度较大，用于快速定位目标区域；底层图节点最为完整，负责最终的精确近邻搜索。因此，HNSW 可以理解为”跳表在高维向量空间中的推广”。

具体而言，对于每一个数据点 x_i，HNSW 都会随机生成一个最大层数 l_i。具体的采样方式如下：

$$l_i = \lfloor -m_L \ln(u) \rfloor \tag{18.155}$$

其中，m_L 是控制分层稀疏程度的超参数，u 是 $[0,1)$ 区间内的均匀分布随机数，即 $u \sim \mathcal{U}(0,1)$。因此，层数 l 服从指数衰减分布：

$$P(L \geq l) = e^{-l/m_L}, \tag{18.156}$$

由于指数分布的特点，大部分节点仅存在于第 0 层，而只有极少数节点能够出现在较高层。因此，第 0 层包含全部数据点；随着层数不断升高，节点数量迅速减少，如图18.16所示。

需要注意的是，这种层次结构通常来说具备两个重要作用。一方面，高层节点数量较少，因此可以快速完成大范围导航，迅速缩小搜索区域；另一方面，当搜索逐渐接近目标之后，再进入更低层进行局部搜索，从而不断提高搜索精度。整个搜索过程会遵循“由粗到细（Coarse-to-Fine）”的思想，与前面介绍的 IVFADC、随机投影

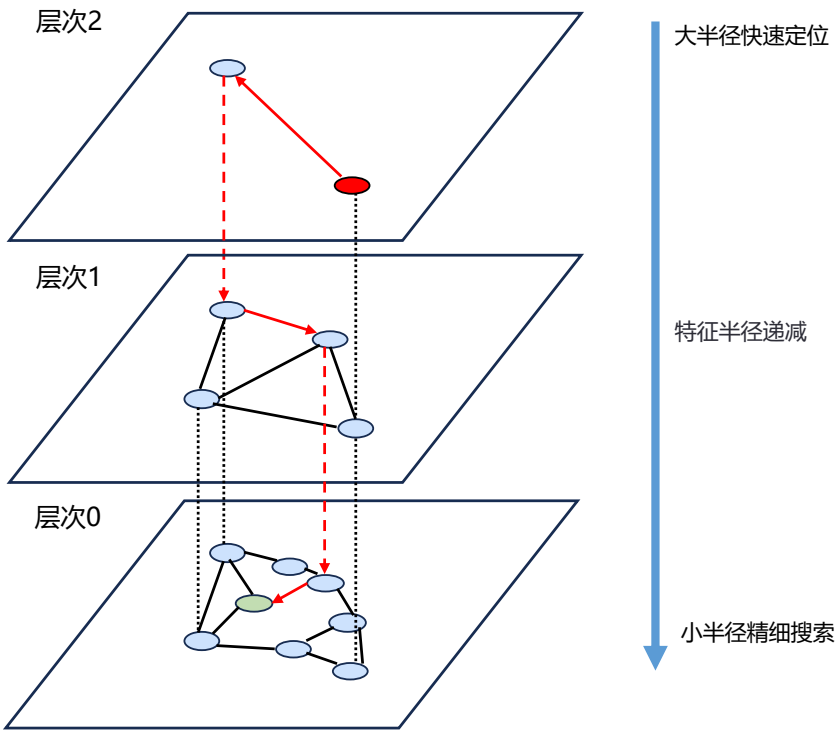

图 18.16: HNSW 多层图结构示意图。高层节点较少但连接跨度较大，底层节点最完整，用于最终近邻搜索。

树逐层划分等方法具有类似的层次搜索思想，但 HNSW 采用的是基于图的导航而非针对高维空间进行划分的方式，因此 HNSW 具有更强的适应复杂数据分布的能力。

此外，需要注意的是，HNSW 并不依赖欧氏空间的几何性质，其整个搜索过程仅依赖距离函数（Distance Function）。因此，无论是欧氏距离（L2 Distance）、余弦距离（Cosine Distance）、内积距离（Inner Product），还是编辑距离（Edit Distance）、Jaccard 距离等非欧度量空间，均可以直接应用 HNSW 构建近似最近邻索引。这也是 HNSW 相比 KD 树、Ball Tree 等传统空间划分方法的一项重要优势。

18.5.1 HNSW 构图过程

HNSW 采用增量式（Incremental）的方式构建整个近邻图。与 KD 树、PQ 等需要离线构建整个索引不同，HNSW 支持逐个向图中插入新的数据点，因此天然支持在线更新（Online Update），也是其能够广泛应用于工业界的重要原因之一。具体而言，对于每一个待插入的数据点 x，HNSW 需要依次完成如下几个步骤：

1. 随机生成节点最高层数；
2. 从最高层开始进行 Greedy Search 寻找距离 x 最近的入口节点；
3. 自顶向下逐层搜索，每层维护一个候选集合；
4. 在每层选择最终保留的邻居节点；
5. 与这些邻居建立双向连接（Bidirectional Link）。

对于每一个新插入的数据点 x，首先随机生成其所在的最高层数 l。l 的采样方式仍然采用前文所述的公式 $l_i = \lfloor -m_L \ln(u) \rfloor$。由于 l 服从指数衰减分布，因此大部分节点仅存在于第 0 层，而只有极少数节点能够进入较高层。例如，当共有 100 万个节点时，第 5 层可能仅包含几百个节点，而最高层甚至只有个位数节点。这种指数衰减结构使得不同层节点数量满足近似几何递减关系，从而保证了高层具有较大的搜索跨度，而底层则能够保留完整的数据分布。

当节点层数确定之后，需要将新节点插入已有图中。假设当前图已经构建完成，并维护一个最高层入口节点（Entry Point）。插入新节点 x 时，首先从最高层入口节点开始进行贪心搜索（Greedy Search）。设当前访问节

点为 v，其邻居集合为

$$\mathcal{N}(v) = \{v_1, v_2, \cdots, v_M\}, \tag{18.157}$$

分别计算所有邻居到查询点 x 的距离 $d(x, v_i)$。若存在某个邻居满足

$$d(x, v_i) < d(x, v) \tag{18.158}$$

则移动到对应邻居继续搜索；否则认为已经到达当前层的局部最优位置，并进入下一层继续搜索。由于高层节点数量较少，因此贪心搜索通常仅需要访问很少数量的节点便能够快速逼近目标区域。当下降到下一层之后，再以当前找到的节点作为新的入口继续搜索。整个搜索过程不断重复，直到第 0 层结束。

完成贪心搜索之后，可以得到当前层的一组候选邻居（Candidate Set）。一种最直接的方法是选择距离最近的 M 个节点建立连接，但论文指出这种策略容易导致图结构退化。如图18.17所示，如果简单选择距离最近的 M 个邻居，则这些邻居往往来自同一个局部簇（Cluster），节点之间高度相似，导致不同簇之间缺乏连接。当搜索进入某一个局部区域之后，很容易陷入局部最优，从而降低整个图的导航能力。HNSW 因此提出了 Neighbor Selection Heuristic 的启发式策略。

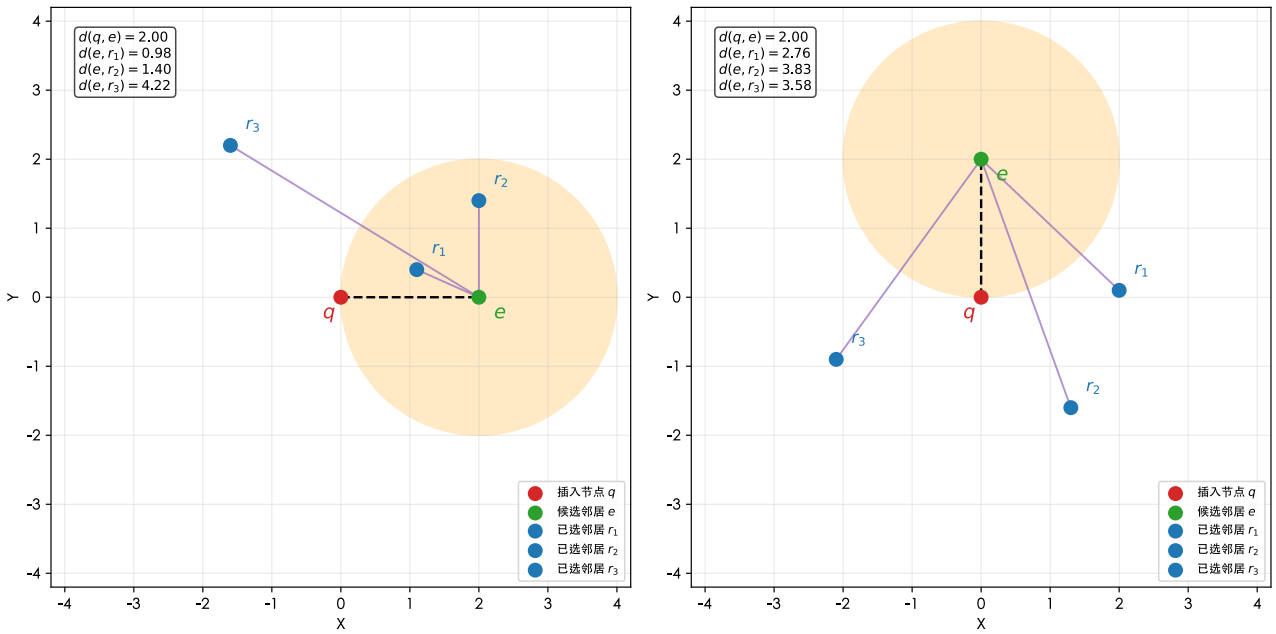

图 18.17: Neighbor Selection Heuristic 示意图。左图中候选邻居 e 会被丢弃，右图中候选邻居 e 会保留。

具体而言，首先按照距离由近到远对候选节点进行排序：$v_1, v_2, \cdots, v_n$，假设最近的候选节点为 v_i，已经保留的邻居集合记为 S。如果满足以下条件

$$d(x, v_i) \le d(v_i, s) \qquad \forall s \in S \tag{18.159}$$

时，才将 v_i 加入最终邻居集合。换句话说，如果某个候选节点已经能够通过已有邻居到达，则无需重复建立连接；只有能够提供新的导航方向时，才保留该连接。最终每一层的邻居保留至多 M 条边：$|S| \le M$。相比简单选择距离最近的 M 个邻居，这种策略能够保留更多方向上的连接，使不同簇之间形成更多跨簇边（Long-range Link），从而显著提升整个图的连通性（Connectivity）和导航能力（Navigability）。从图18.17也能看出，使用该条件之后，右图选出来的邻居 e 与其他已经选择的邻居 r_i 并没有存在局部扎堆的情况，反而是拓宽了新的邻居方向。

在完成邻居选择之后，新节点与所有保留邻居建立双向连接，同时若某个邻居的连接数超过参数 M，则再次利用 Neighbor Selection Heuristic 删除冗余边，使每个节点始终保持有限数量的邻居。整个构图过程按照数据插入顺序不断重复，最终形成一张具有良好导航性质的小世界图。由于每次插入仅访问局部节点，因此论文证明，在均匀数据分布假设下，HNSW 整体构图复杂度约为 $\mathcal{O}(N \log N)$，远优于传统近邻图需要两两计算距离所

带来的 $\mathcal{O}(N^2)$ 构图复杂度，因此能够支持百万乃至上亿规模向量数据的索引构建。

18.5.2 HNSW 查询过程

HNSW 的查询过程与构图阶段的搜索过程类似，同样采用自顶向下（Top-down）的层次化搜索策略，但底层搜索更加充分，其核心由参数 $efSearch$ 控制。设查询向量为 q，HNSW 首先从最高层（Level $L_{\max}$）的入口节点（Entry Point）开始搜索。在高层图中，由于节点数量较少，因此采用贪心搜索（Greedy Search）即可快速定位到距离 q 最近的局部区域。

具体来说，设当前节点为 v，首先计算其与查询向量之间的距离 $d(q,v)$，然后遍历 v 所有邻居。如果存在某个邻居距离 q 更近，则移动到该邻居继续搜索；否则认为已经到达当前层的局部最优节点，并下降到下一层继续搜索。由于高层图十分稀疏，因此每层通常只需访问极少量节点即可完成定位，其主要作用是快速缩小搜索范围，而不是寻找最终最近邻。不断重复上述过程，直到到达最底层（Level 0）。

在底层搜索则采用与高层不同的搜索策略。由于底层图连接最为稠密，仅依赖贪心搜索容易陷入局部最优，因此 HNSW 在底层引入了 Best-First Search 策略。该算法首先会维护两个优先队列，即候选队列 (Candidate Queue) 与结果队列 (Result Queue)。其中候选队列按照距离从近到远排序，保存下一步需要继续扩展的候选节点；而结果队列则保存目前已经找到的距离最近的 $efSearch$ 个节点。在初始化时，通常会将高层搜索得到的入口节点同时放入两个队列。随后不断地重复以下的步骤：

1. 从候选队列中取出当前距离查询最近的节点 v；
2. 遍历 v 的所有邻居；
3. 若某邻居尚未访问，则计算其与 q 之间的距离；
4. 若距离优于结果队列中的最差节点，则加入两个队列；
5. 更新结果队列，仅保留距离最近的 $efSearch$ 个候选节点。

整个搜索过程持续进行，直到候选队列中距离最近的节点已经不可能优于结果队列中的最差节点，此时算法停止搜索。最终，再从 Result Queue 中返回距离最近的 Top-K 个结果作为最终近邻。由于搜索过程中允许同时保留多个候选路径，因此相比于简单贪心搜索，Best-First Search 能够有效跳出局部最优，提高近邻搜索的召回率。

实际上，HNSW 可以看作是在高层采用贪心搜索进行快速导航，而在底层采用 Best-First Search 进行局部精细搜索，两者共同完成整个 ANN 查询过程。其中，高层主要负责快速缩小搜索空间，底层则负责最终的近邻搜索，因此底层搜索质量直接决定了整个 HNSW 的检索精度。

18.5.3 HNSW 复杂度与超参数分析

HNSW 之所以能够在工业界得到广泛应用，一个重要原因在于其构图和查询复杂度均接近对数级，并且能够通过少量超参数灵活地平衡召回率、查询延迟以及索引规模。首先我们分析一下 HNSW 的索引构建过程。假设数据库包含 N 个向量，每插入一个新的节点，都需要首先利用当前已经构建好的 HNSW 图执行一次贪心搜索，从最高层逐层往下搜索，最终在底层找到距离最近的候选邻居。随后，再利用 Neighbor Selection Heuristic 算法从候选集中选择最终连接的邻居节点，并建立双向边。因此，一个节点的插入复杂度主要由图搜索产生。

由于 HNSW 采用指数衰减的层数分布，高层节点数量随着层数指数下降，因此搜索路径长度近似满足对数增长：

$$L_{\max} \approx \log_{1/p} N, \tag{18.160}$$

其中 p 表示节点继续进入更高层的概率。因此，每插入一个节点仅需要访问少量节点即可完成定位，其平均复杂度约为 $\mathcal{O}(\log N)$。而整个数据集共包含 N 个节点，因此 HNSW 整体构图复杂度约为 $\mathcal{O}(N \log N)$。相比于需要计算全部两两距离的 KNN 图的构建方法，其复杂度由 $\mathcal{O}(N^2)$ 下降到了近似线性增长，因此能够支持百万乃至千万级向量索引构建。

接下来我们分析查询的复杂度。对于一个新的查询向量 q，HNSW 首先从最高层入口节点开始执行贪心搜

索，每一层只需要访问极少量的节点就可以快速逼近目标区域，然后逐层下降，最终在底层执行较大范围的局部搜索得到最终结果。由于图的层数满足 $L_{\max} = O(\log N)$，且每层访问节点数量通常保持为一个常数，因此理论上的平均查询复杂度同样近似为 $\mathcal{O}(\log N)$。

当然，这只是理论平均复杂度。在实际工程中，底层通常需要维护一个候选队列，因此真正访问的节点数量主要由参数 $efSearch$ 决定。因此，更准确地说，一次查询大约访问 $O(efSearch)$ 个节点，而由于搜索路径长度随着图规模近似呈对数增长，因此总体复杂度通常写作 $\mathcal{O}(efSearch \cdot \log N)$。由于 $efSearch$ 一般仅设置为几十到几百，因此即使数据库规模达到数亿甚至十亿级，查询速度依然能够保持毫秒级。

然后，我们分析一下 HNSW 中的关键超参数以及它们的作用。HNSW 主要包含了三个关键的超参数，它们共同决定了索引规模、构建速度以及最终召回率。其中，参数 M 表示每个节点最多保留多少条邻居边，也是 HNSW 中最重要的参数之一。M 越大，每个节点连接的邻居越多，整个图会更加稠密，因此，图的连通性会更好、查询路径更加丰富、召回率通常更高、内存占用也随之增加、构图时间也会更长。在工业实践中，通常选取 $M \in [16, 64]$。而 Faiss 默认一般采用 $M = 32$ 的取值。

超参数 $efConstruction$ 表示构图过程中贪心搜素需要维护的候选集合大小。我们可以理解为：构图时允许搜索得有多仔细。$efConstruction$ 越大，表明能够找到真正近邻的概率越高、图结构质量越好、召回率更高、但是构图速度也会明显下降。该参数通常满足 $efConstruction > M$ 的条件。在工业实践中，我们一般设置 $efConstruction = 100 \sim 400$。$efSearch$ 表示查询过程中维护的候选队列大小。我们可以理解为该参数决定了搜索过程中允许探索多少个候选节点。如果 $efSearch$ 越大，那么说明搜索更加充分、召回率不断提高、查询延迟也会同步增加。$efSearch$ 通常满足 $efSearch \geq K$ 的条件，其中 K 表示最终返回的 Top-K 结果。工业实践中通常取 $efSearch = 50 \sim 500$。下表18.2展示了这三个超参数分别对于召回率、查询耗时等指标的影响。

表 18.2: HNSW 三个关键超参数的作用。

参数	召回率	查询耗时	内存开销	构图时间
M	显著提高	略有增加	显著增加	略有增加
$efConstruction$	提高	无影响	无影响	显著增加
$efSearch$	显著提高	显著增加	无影响	无影响

从表中可以看出：增大 M 主要能够提高图质量，但会增加内存开销；而增大 $efConstruction$ 主要会提高构图质量、增大 $efSearch$ 主要提高查询召回率。因此，在实际部署过程中通常首先固定 M，然后离线调节 $efConstruction$ 保证图质量，最后在线动态调整 $efSearch$ 来平衡召回率与查询延迟。

18.5.4 Faiss 中的 HNSW 实现

目前 HNSW 已经成为工业界应用最广泛的 ANN 算法之一，Faiss、Milvus、Weaviate、Qdrant、Vespa 等主流向量数据库均提供了 HNSW 索引实现。其中，Faiss 主要提供以下几种 HNSW 索引：

- **IndexHNSWFlat**：底层直接存储原始浮点向量，距离计算精确，召回率最高，但内存占用较大；
- **IndexHNSWPQ**：底层采用 PQ 压缩向量，利用 ADC 完成距离计算，在降低内存占用的同时保持较高召回率；
- **IndexHNSWSQ**：结合 Scalar Quantization 进一步压缩存储空间；
- **IVF+HNSW**：利用 HNSW 替代传统 IVF 中的粗聚类搜索，提高粗召回阶段效率。

近年来，大多数工业级向量数据库均默认采用 HNSW 作为核心索引结构。例如，Milvus 默认使用 HNSW 作为浮点向量索引；Qdrant 几乎所有向量检索均建立在 HNSW 之上；Weaviate 同样采用 HNSW 构建向量图；而 Faiss 则提供了 HNSW 与 PQ、IVF 等多种索引结构的灵活组合，使用户能够根据内存预算、查询延迟以及召回率要求自由选择最适合的索引方案。

总体来看，HNSW 代表了当前图索引路线中最成熟、最稳定、工程实践最广泛的一类 ANN 算法。相比 KD-Tree、Ball Tree 等传统空间划分方法，HNSW 几乎不受维度灾难影响；相比 LSH 等概率哈希方法，其召回率更

高；相比 PQ 等压缩方法，其能够直接作用于原始向量空间而无需量化误差。因此，在当前主流推荐系统、向量数据库以及大语言模型 RAG 系统中，HNSW 已经成为最常用的近似最近邻检索算法之一。

18.6 向量检索工具

前面章节主要介绍了近似最近邻（Approximate Nearest Neighbor，ANN）搜索中的经典算法，包括 KD-Tree、Ball Tree、随机投影树（RP-Tree）、局部敏感哈希（LSH）、乘积量化（PQ）以及 HNSW 等。这些算法为高维向量检索提供了理论基础。然而，在实际工业系统中，开发者通常不会直接实现这些底层算法，而是借助成熟的向量检索框架或向量数据库完成索引构建、查询服务以及集群部署。

目前工业界较为主流的向量检索工具包括 Faiss、ScaNN、DiskANN 等 ANN 检索框架，以及 Milvus、Qdrant、Weaviate、Pinecone、Elasticsearch Vector Search 等向量数据库。其中，Faiss 是目前应用最广泛的 ANN 算法库；ScaNN 针对 CPU 环境进行了大量优化；DiskANN 则面向十亿级甚至百亿级向量检索场景。而 Milvus、Qdrant 等向量数据库则进一步提供了索引管理、数据持久化、分布式部署以及高可用等完整的工程能力。

下面分别对这些代表性的向量检索工具进行介绍。Faiss（Facebook AI Similarity Search）是 Meta 开源的高性能向量检索库，也是目前工业界应用最广泛的 ANN 框架之一。Faiss 几乎集成了目前主流的 ANN 算法，包括暴力搜索、IVF、PQ、HNSW、LSH 等，并支持 CPU 和 GPU 两种计算平台。Faiss 内部包含丰富的索引类型，包括 IndexFlat、IndexIVF、IndexPQ、IndexHNSW、IndexLSH、GPU Index 等。其中，各类索引的特点如下：

- **IndexFlat**：暴力搜索，不进行任何索引，召回率最高，但计算复杂度为 $\mathcal{O}(ND)$；
- **IndexIVFFlat**：采用 IVF 倒排索引缩小搜索范围，每个倒排桶内仍采用精确距离计算；
- **IndexIVFPQ**：IVF 结合 PQ 压缩，是目前工业界应用最广泛的索引结构之一；
- **IndexPQ**：直接采用 PQ 压缩全部向量，不使用倒排索引；
- **IndexHNSW**：基于 HNSW 图索引进行 ANN 搜索，具有较高召回率；
- **IndexLSH**：采用随机超平面 LSH 实现向量哈希检索。

在具体实践时，用户可以根据数据规模、内存预算以及召回率要求选择最适合的索引结构。例如，IndexFlat 适用于小规模数据集，提供最高召回率但计算复杂度较高；IndexIVFPQ 结合倒排索引与乘积量化，适用于中等规模数据集；IndexHNSW 则在大规模数据集上提供高召回率和低延迟。

此外，Faiss 库统一采用“训练（Train）- 建库（Add）—查询（Search）”这种三阶段的接口。首先，对于 IVF、PQ 等需要学习码本或聚类中心的索引，需要调用 `train()` 完成离线训练；随后调用 `add()` 将数据库向量加入索引；在线查询阶段则调用 `search()` 返回 Top-K 近邻。Faiss 还支持索引合并（Merge）、删除（Remove）、GPU 索引、分片（Shard）以及副本（Replica）等工程能力。其中，Shard 将整个向量库划分为多个分片分别存储，而 Replica 则构建多个相同索引副本以提高查询吞吐，因此 Faiss 不仅适用于单机实验，也能够支撑较大规模的在线向量检索服务。

ScaNN（Scalable Nearest Neighbors）是 Google 提出的一种高性能 ANN 检索框架，主要针对 CPU 环境进行了大量优化，目前也是 Google 内部推荐系统和搜索系统的重要基础组件之一。相比 Faiss 主要围绕 PQ 展开优化，ScaNN 提出了更加适合向量检索任务的各向异性向量量化（Anisotropic Vector Quantization，AVQ）方法。AVQ 不再简单最小化欧氏距离，而是重点降低影响内积排序误差的方向量化误差，因此在推荐系统的 Embedding 检索任务中通常具有更高的召回率。ScaNN 整体检索流程主要包含三个阶段：

- Tree Partition：首先利用聚类树对整个向量空间进行粗划分；
- AVQ Quantization：对子空间进行各向异性向量量化；
- Reordering：对候选集合重新计算精确距离完成最终排序。

除此之外，ScaNN 还针对现代 CPU 进行了 SIMD 向量化、Cache 优化以及 TPU 友好的矩阵计算优化，因此在 Google 公开实验中，相较传统 PQ 方法通常能够获得更高的查询速度和 Recall。

DiskANN 是微软于 2019 年提出的大规模 ANN 检索系统，其最大的特点是在保持较高 Recall 的同时，将大部分索引存储于 SSD 中，仅保留少量热点节点驻留内存，因此能够突破传统内存容量限制，实现十亿级甚至百亿级向量检索。DiskANN 整体仍然采用图索引结构，但进一步引入了内存与磁盘协同设计。通常情况下，热点

节点（Cache Nodes）保存在 RAM 中；完整图结构存储于 SSD；查询过程中优先访问内存，再异步读取 SSD 节点。这种设计有效降低了内存占用，同时利用现代 NVMe SSD 较高的随机读性能，使得查询延迟仍然保持在毫秒级。目前 DiskANN 已经广泛应用于微软 Bing 搜索、Azure AI Search 以及许多超大规模 RAG 系统中，也是当前工业界超大规模向量检索的重要方案之一。

随着生成式 AI、大语言模型以及 RAG（Retrieval-Augmented Generation）技术的发展，仅提供 ANN 索引已经无法满足实际业务需求，因此近年来出现了大量面向向量检索的数据库系统（Vector Database）。向量数据库不仅提供 ANN 索引，还集成了数据持久化、事务管理、分布式部署、多租户管理、过滤查询以及 RESTful API 等完整的数据管理能力。目前较为主流的向量数据库包括：

- **Milvus**：目前应用最广泛的开源向量数据库之一，支持 HNSW、IVF、PQ、DiskANN 等多种索引结构，具有良好的分布式扩展能力；
- **Qdrant**：基于 Rust 开发，默认采用 HNSW 作为底层索引，支持 Payload 过滤，非常适合 RAG 系统；
- **Weaviate**：强调知识图谱与向量检索融合，提供丰富的 Schema 管理能力；
- **Pinecone**：商业化 SaaS 向量数据库，无需用户自行部署，适用于云端 AI 应用；
- **Elasticsearch Vector Search**：在传统倒排索引基础上增加向量检索能力，将全文检索与 ANN 搜索进行了统一，非常适合混合检索（Hybrid Search）场景。

总体来看，目前工业界的向量检索系统已经逐渐形成了较为清晰的技术路线：Faiss、ScaNN 以及 DiskANN 等框架主要负责底层 ANN 索引算法的实现，而 Milvus、Qdrant、Weaviate、Pinecone 以及 Elasticsearch 等向量数据库则进一步提供完整的数据管理、分布式部署以及在线服务能力，成为当前大模型 RAG 系统、推荐系统召回以及搜索引擎的重要基础设施。

18.7 向量召回系统架构

前面几节介绍了 KD-Tree、LSH、PQ、HNSW 等近似最近邻（Approximate Nearest Neighbor，ANN）算法。这些算法解决的是“如何快速查找相似向量”的问题，而在真实工业系统中，仅有 ANN 算法本身远远不够，还需要围绕模型训练、Embedding 生成、索引构建、在线查询以及索引更新等环节构建完整的向量检索系统。

这里我们以推荐系统中最经典的 User-to-Item（U2I）召回为例，对工业级向量检索系统的整体架构进行介绍，如图18.18所示。整个系统通常由离线索引构建、在线 Embedding 生成以及 ANN 检索服务三部分组成。其中，离线部分负责生成物品向量并构建 ANN 索引；在线部分负责实时计算用户向量，并调用 ANN 服务完成候选召回。

18.7.0.0.1 离线 Embedding 生成与索引构建 推荐模型训练完成之后，通常会得到用户塔（User Tower）和物品塔（Item Tower）两个编码网络。其中，物品数量虽然巨大，但更新频率相对较低，因此工业界通常采用离线方式计算所有物品 Embedding。离线 Runner（通常采用 C++ Runner 或者 Java Runner）会周期性扫描全量物品，调用物品塔进行前向推理，生成最新的 Item Embedding，并写入 Embedding Server 或者对象存储中。随后，索引构建服务读取最新 Embedding，进而根据业务需求构建对应的 ANN 索引，例如 HNSW、IVF、IVF+PQ 或者 DiskANN 等。

为了保证线上服务的稳定性，ANN 索引通常采用离线异步构建的方式。当新的索引构建完成后，再通过**原子切换（Atomic Switch）**或者**双缓冲（Double Buffer）机制**完成索引的热更新，从而避免索引重建过程影响线上查询服务。其中，双缓冲（Double Buffer）是工业界推荐系统中最常用的索引更新方案。其基本思想如下：

定义 18.12

双缓存的索引构建方式需要在系统中同时维护两份索引：一份作为当前线上索引持续提供 ANN 查询服务；另一份则在后台完成新索引的构建。当后台索引构建完成并经过完整校验后，再通过一次原子切换，将线上服务切换到新的索引，而旧索引则被释放或保留一段时间作为回滚备份。整个更新过程无需停止线上服务，因此能够实现索引的无缝热更新。

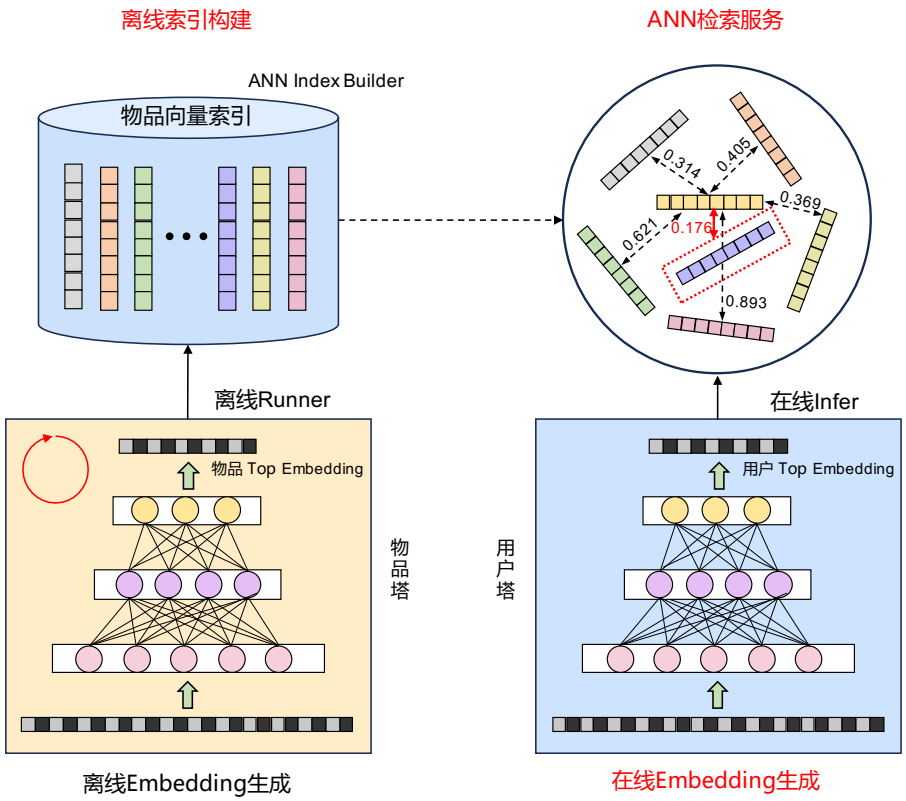

图 18.18: 工业级向量检索系统整体架构。

不过，双缓冲方案在底层实现时通常依赖多线程并发机制，因此需要特别关注线程安全问题。例如，在索引切换过程中，通常需要借助互斥锁（Mutex）、读写锁（Read-Write Lock）或者原子变量（Atomic Variable）等同步机制，保证索引切换操作的原子性，避免查询线程访问到正在构建或者已经释放的索引对象。

除了线程安全之外，更容易被忽视的是**索引一致性（Index Consistency）问题**。在工业实践中，新索引的构建通常涉及离线构建线程、增量更新线程以及线上查询线程等多个模块协同工作。如果索引构建流程设计不完善，就可能导致新旧索引之间出现数据不一致。例如，在全量索引构建过程中，部分 Item 由于线程竞争（Race Condition）、任务异常退出、数据同步失败等原因未能成功写入新索引；又或者在后台构建新索引期间，新的 Item 持续写入旧索引，而这些增量数据未能正确同步或回放（Replay）到新索引。当后台索引构建完成后，如果没有经过严格的一致性校验便直接完成索引切换，就可能导致部分候选 Item 永久缺失。

这类问题在线上通常表现为候选集合数量异常。例如，在某些业务场景中，整个候选集合实际上只有约 100 个 Item，此时即使采用全量 ANN 召回，也理应返回全部候选 Item。然而，实际线上返回的候选数量却明显少于 100 个。由于 ANN 召回本身没有任何过滤逻辑，因此，这种现象往往并非 ANN 算法本身造成的召回损失，而是索引构建或索引切换过程中部分 Item 未能正确进入 ANN 索引所导致的。实际工程排查中，这类问题最终通常定位到双缓冲索引构建流程，而非 ANN 搜索算法本身。

相比之下，在拥有数千万甚至数十亿候选物品的大规模 ANN 系统中，这类问题反而更加隐蔽。由于候选集合规模巨大，即使少量 Item 没有成功写入索引，对最终 Top-K 召回结果以及线上 A/B 实验指标的影响通常也十分有限，因此业务指标往往不会出现明显异常。这种现象容易掩盖底层索引构建存在的问题，使得 Bug 长期潜伏在线上系统之中，直到某些特殊业务场景或者离线校验时才被发现。

因此，在工业级 ANN 系统开发过程中，除了关注索引构建速度和查询性能之外，还需要重点保证整个索引生命周期（Index Lifecycle）的正确性。通常需要建立完善的索引完整性校验机制，例如索引中 Item 数量校验、离线 Embedding 数量与索引数量一致性校验、增量数据回放校验、索引切换后的随机抽样校验以及线上召回覆盖率监控等，从而保证索引构建、索引更新以及索引切换过程的完整性与一致性。实践经验表明，当线上 ANN 召回率出现异常下降时，应首先检查索引构建与索引更新流程是否存在问题，而不是立即怀疑 ANN 算法本身，

因为许多看似“算法召回率下降”的问题，本质上往往是索引构建异常所导致的工程问题。

18.7.0.0.2 在线 Embedding 生成 相比物品 Embedding 可以提前离线计算，用户 Embedding 则需要根据用户当前状态实时生成，因此在线 Embedding 生成是整个向量召回链路中的关键环节。当用户请求到达推荐系统之后，首先会从 Feature Server、Embedding Server 或者特征缓存（Feature Cache）中获取用户历史行为序列、用户画像、上下文特征以及各类 Embedding 特征，然后送入用户塔（User Tower）进行一次前向推理，得到当前用户对应的 Embedding 向量：

$$\mathbf{q} = f_{\text{UserTower}}(x_u) \tag{18.161}$$

其中，x_u 表示用户实时特征，$\mathbf{q}$ 即为本次 ANN 检索所使用的 Query Embedding。随后，该 Embedding 向量将作为查询 Key 发送至 ANN 检索服务，在海量物品 Embedding 中完成近似最近邻搜索，返回 Top-K 候选物品。

由于在线 Embedding 的生成需要在每次用户请求中实时完成，因此其计算延迟会直接影响整个召回阶段的响应时间。一般来说，推荐系统从请求接收到完成召回通常只有几十毫秒的延迟预算，而用户塔推理往往占据其中的重要组成部分。因此，用户塔通常部署在高性能在线推理服务中，并结合 TensorRT、ONNX Runtime、TorchScript 等推理引擎进行模型优化，同时配合 FP16 量化、算子融合（Operator Fusion）、Batch 推理以及 SIMD/GPU 加速等技术，尽可能降低单次推理延迟，提高系统吞吐量。相比之下，物品 Embedding 通常采用离线批量计算并缓存到 Embedding Server 或者 ANN 索引中，因此其计算开销不会出现在在线请求路径上，对整体响应时间的影响较小。

在工业实践中，除了模型推理本身之外，特征获取往往也是影响在线时延的重要因素。为了降低网络访问开销，用户画像、统计特征以及 Embedding 特征通常都会缓存在高性能 KV 存储或者内存数据库中，尽量减少远程 RPC 调用次数。同时，一些变化较慢的用户 Embedding 还会采用短时间缓存（Short-term Cache）策略，在用户连续请求期间直接复用最近一次计算得到的 Embedding，仅当用户行为发生明显变化或者缓存失效时才重新执行模型推理，从而进一步降低整体计算开销。通常这一类缓存策略都会集成进在线推理服务中，当服务整体面临比较大的耗时压力时可以开启缓存策略降低调用 Embedding Server 的耗时，而在追求最高召回率的场景下则可以关闭缓存，保证每次请求都使用最新的用户 Embedding。

此外，一个推荐系统通常会部署多路召回模型，例如 U2I 召回、Graph 召回、标签召回、多兴趣召回（Multi-Interest Retrieval）以及双向召回等。不同召回模型通常拥有各自独立的用户塔，并生成不同语义空间下的用户 Embedding，以满足不同召回策略的需求。然而，这些模型往往共享大量相同的输入特征，例如用户画像、历史行为序列以及上下文信息。如果每一路召回都独立完成一次特征读取和模型推理，将造成大量重复计算和资源浪费。因此，在实际工程中，通常会将多个用户塔统一部署到同一个在线推理服务中，共享底层特征处理流程，并采用多任务推理（Multi-task Inference）或者共享 Backbone、独立 Head 的模型结构，一次前向计算即可同时输出多个召回模型所需的用户 Embedding，从而减少重复计算和 RPC 调用次数，降低整体系统延迟，提高 GPU 利用率和系统吞吐能力。

进一步地，在许多工业系统中，多路召回模型的用户塔网络结构往往完全一致，仅物品塔或者后续的 ANN 索引有所不同。例如，不同召回服务可能针对不同的候选物品集合分别构建 ANN 索引，但用户侧 Embedding 空间保持一致。在这种情况下，用户 Embedding 仅需在线计算一次，随后即可复用于多个召回服务，而无需针对每一路召回重复执行用户塔前向推理。多个 ANN 召回服务共享同一个查询 Embedding，不仅能够进一步降低在线推理开销，还能够减少 GPU 计算资源消耗，因此也是工业级推荐系统中一种十分常见的优化方案。

18.7.0.0.3 ANN 向量检索服务 得到用户 Embedding 之后，召回服务会将其作为查询向量发送至 ANN 检索服务。ANN 服务通常部署在高性能 RPC 服务中，内部维护着海量物品 Embedding 构建出的索引结构，例如 HNSW、IVF+PQ 或者 DiskANN 等。根据用户向量快速完成近似最近邻搜索，返回 Top-K 个最相似的物品 ID：

$$\text{TopK} = \text{ANNSearch}(\mathbf{q}, \mathcal{I}) \tag{18.162}$$

其中 $\mathcal{I}$ 表示 ANN 索引。这一过程实际上并不会直接返回最终推荐结果，而是返回当前一路召回对应的候选集合（Candidate Set）。后续这些候选物品还需要进入粗排、精排以及重排等排序阶段进一步计算各种预测指标，最终生成推荐列表。由于 ANN 算法仅完成向量近邻搜索，因此整个召回阶段通常存在多路召回，例如双塔召回、Graph 召回、标签召回、协同过滤召回等，不同召回源返回的候选集合最终会统一汇总进入排序阶段。

由于工业系统中的物品不断新增、删除或者发生内容更新，因此 ANN 服务的索引也需要持续更新。一种简单的方法是周期性全量重建索引，例如每天或每小时重新计算全部 Embedding 并重新构建 ANN 索引。这种方式实现简单，适用于更新频率较低的业务场景。对于短视频、新闻推荐、电商等内容更新速度较快的场景，则通常采用“增量更新 + 周期重建”相结合的方式。新增物品首先通过增量接口插入 ANN 索引，当累计更新达到一定规模之后，再统一进行一次全量索引重建，以保证索引质量和查询性能之间取得平衡。对于 HNSW 等动态图结构，还支持在线插入（Insert）操作，因此能够较好支持实时更新；而 IVF+PQ 等索引由于涉及聚类中心和量化码本，通常仍然采用周期性重建策略。

18.7.0.0.4　分布式 ANN 服务　随着物品规模不断扩大，单台服务器已经无法存储全部 Embedding 和 ANN 索引，因此工业界通常采用分布式部署方式。最常见的方法是按照物品 ID 或者 Embedding 进行水平切分（Shard），每个 Shard 维护自己对应的 ANN 索引。在线查询时，用户请求会在 ANN 检索服务内部路由到多个不同的 ANN Shard，每个 Shard 分别完成局部 Top-K 检索：

$$\{\text{TopK}_1, \text{TopK}_2, \cdots, \text{TopK}_S\}, \tag{18.163}$$

其中 S 表示 Shard 数量。随后，Aggregator 负责汇总所有 Shard 返回结果，并重新排序得到最终 Top-K 候选：

$$\text{TopK} = \text{Merge}(\text{TopK}_1, \cdots, \text{TopK}_S) \tag{18.164}$$

为了保证高可用性，每个 Shard 通常都会部署多个 Replica 副本，通过一致性哈希、负载均衡以及故障自动切换等机制保证线上服务稳定运行。

目前，工业界的 ANN 服务大多采用 Faiss、ScaNN、DiskANN 等高性能检索库作为底层实现，并结合 RPC 服务封装统一的 ANN 接口，对外提供 Batch Search、Top-K Search、Range Search 等能力。为了进一步提升整个 ANN 服务的吞吐量，ANN 服务通常还会结合 SIMD 向量化指令、GPU 加速、多线程并行计算、内存映射（Memory Mapping）、NUMA 优化以及查询批处理（Batch Query）等工程优化技术，使得单台服务器即可支持数百万 QPS 级别的向量距离计算。

总体来看，一个完整的工业级向量检索系统实际上涵盖了模型训练、Embedding 生成、索引构建、索引更新、在线 ANN 查询、分布式部署以及工程优化等多个模块。其中，ANN 算法只是整个系统中的核心组件之一，而真正决定线上性能的往往是整个系统架构的设计。随着生成式推荐、RAG 以及 Agent 等技术的发展，向量检索系统已经成为现代推荐系统与大模型系统中的基础设施之一，也是连接 Embedding 模型与下游业务的重要桥梁。

18.8 本章小结

本章节围绕工业级推荐系统中的向量检索 (Vector Retrieval) 展开介绍，系统梳理了当前主流的近似最近邻 (Approximate Nearest Neighbor，ANN) 检索算法、工业级向量检索框架以及完整的向量检索系统架构。从底层算法原理到工程实现，再到线上系统部署，希望能够帮助读者建立从理论到实践的完整知识体系。

首先，我们按照底层数据结构的不同，将 ANN 检索算法划分为基于树、基于哈希、基于量化以及基于图四大类技术路线，并重点分析了每一类算法提出的背景、核心思想以及所解决的问题。在基于树的方法中，我们介绍了 KD-Tree、Ball Tree 以及随机投影树 (Random Projection Tree，RP-Tree)。其中，KD 树通过轴对齐划分空间实现高效搜索，但在高维空间容易受到维度灾难 (Curse of Dimensionality) 的影响；Ball Tree 采用超球体划分方式，在一定程度上缓解了 KD 树轴对齐划分带来的不足，但仍然存在球体重叠导致搜索效率下降的问题；随机投影树则利用随机超平面对空间进行递归划分，不仅更加适合高维数据，还具有良好的理论保证。针对随机投影树，我们进一步介绍了 Johnson–Lindenstrauss(JL) 引理的发展背景及其证明思路，并结合 Annoy 框架分析了随机投影树在工业界中的具体实现。

随后，我们介绍了基于局部敏感哈希 (Locality Sensitive Hashing，LSH) 的 ANN 算法。LSH 的核心思想是在保持局部相似性的前提下，将相似对象以较高概率映射到同一哈希桶中，从而避免全量距离计算。围绕这一思想，我们分别介绍了面向集合 Jaccard 相似度的 MinHash 算法以及面向向量余弦相似度的 SimHash 算法，并分析了两者各自适用的应用场景。此外，我们还详细介绍了 LSH Banding 技术，推导了 Banding 碰撞概率公式，分析了 Band 长度 r 与 Band 数量 b 两个关键超参数对于召回率 (Recall) 和精确率 (Precision) 的影响，并结合 Google SimHash 论文介绍了基于倒排索引的海量近重复文档快速检索方案。

接着，我们介绍了基于向量量化 (Vector Quantization) 的乘积量化 (Product Quantization，PQ) 算法。相比树结构和哈希方法，PQ 从向量压缩的角度解决高维检索问题，通过将高维空间划分为多个低维子空间并分别进行量化，实现了指数级码本容量与线性存储开销之间的平衡。我们重点介绍了 PQ 的编码过程以及最核心的对称距离计算 (Symmetric Distance Computation，SDC) 和非对称距离计算 (Asymmetric Distance Computation，ADC) 两种距离估计方法，并分析了查找表如何显著降低距离计算复杂度。随后，我们进一步介绍了 PQ 与倒排索引结合形成的 IVFADC 框架，以及利用残差量化 (Residual Quantization) 进一步降低量化误差的方法。同时，本章还结合 Faiss 框架分析了 PQ 系列算法在工业界中的实际实现。

随后，我们介绍了近年来应用最广泛的图索引算法 HNSW(Hierarchical Navigable Small World)。HNSW 融合了可导航小世界网络 (NSW) 与跳表 (Skip List) 的思想，通过构建多层近邻图，实现了从全局粗搜索到局部精搜索的层次化导航过程。我们详细介绍了 HNSW 的图构建流程、Neighbor Selection Heuristic 邻居选择策略以及贪心搜索过程，并分析了参数 M、$efConstruction$ 和 $efSearch$ 分别对索引规模、构图质量、召回率以及查询延迟的影响。从复杂度角度来看，HNSW 通常具有 $\mathcal{O}(N \log N)$ 的构图复杂度以及平均 $\mathcal{O}(\log N)$ 的查询复杂度，在保证高召回率的同时仍具有较高的查询效率，因此目前已成为工业界应用最广泛的 ANN 算法之一。

在介绍完 ANN 算法之后，我们进一步介绍了工业界常见的向量检索工具，包括 Faiss、ScaNN、DiskANN 等 ANN 检索框架，以及 Milvus、Qdrant、Weaviate、Pinecone、Elasticsearch Vector Search 等向量数据库。其中，Faiss 提供了最为丰富的索引家族，包括 Flat、IVF、PQ、HNSW 以及 LSH 等多种索引结构；ScaNN 通过各向异性向量量化 (Anisotropic Vector Quantization，AVQ) 等技术进一步提升 CPU 环境下的检索性能；DiskANN 则利用 SSD 与内存协同存储，实现了十亿级甚至更大规模向量数据的高效检索，目前已广泛应用于搜索引擎、推荐系统以及 RAG 等场景。

最后，我们从工程实现角度介绍了工业级向量检索系统的整体架构，包括离线 Embedding 生成、离线索引构建、在线 Embedding 生成以及 ANN 检索服务等核心模块。我们分析了双塔召回模型中用户 Embedding 在线生成、物品 Embedding 离线构建的整体流程，并介绍了 Embedding Server、Feature Server 以及 ANN Service 之间的协同工作方式。同时，还讨论了 ANN 索引服务的离线构建、双缓冲 (Double Buffer) 热更新、原子切换 (Atomic Switch) 以及索引一致性等工程问题，并结合实际工业案例分析了双缓冲索引切换过程中可能出现的线程安全、索引丢失以及候选集异常等典型问题。

总体来看，本章节介绍的各种 ANN 算法分别代表了当前工业界近似最近邻检索的几条主要技术路线：KD 树、球树和随机投影树代表了基于空间划分 (Space Partition) 的检索方法；LSH 代表了基于概率哈希 (Hash-based) 的检索方法；PQ 代表了基于向量量化 (Quantization-based) 的检索方法；HNSW 则代表了基于近邻图 (Graph-based) 的检索方法。不同技术路线各具特点，也分别适用于不同的数据规模、硬件资源以及业务场景。在实际工业系统中，这些算法往往并不是孤立使用，而是与倒排索引、向量量化、图索引以及分布式系统等技术相互结合，共同构建出能够支撑亿级甚至百亿级向量检索的工业级 ANN 服务。

本章节也是《推荐系统：工业架构与核心算法》上册的最后一个章节。回顾上册的内容，我们始终遵循”从整体到局部、从架构到算法、从理论到工程”的组织思路，系统介绍了工业级推荐系统的整体架构、核心模块以及典型工程实践。从推荐系统整体架构开始，我们依次介绍了召回、粗排、精排、重排等核心模块的设计原理与实现流程，并结合各阶段经典算法的发展历程，分析了不同算法产生的背景、适用场景以及工程实践中的优化思路。同时，结合大量工业案例，对实际系统开发过程中可能遇到的性能优化、系统稳定性以及工程实现等问题进行了深入讨论。此外，为了紧跟近年来推荐系统的发展趋势，上册还专门增加了生成式推荐相关内容，对生成式推荐的整体架构、关键技术以及工业界最新的研究进展进行了较为系统的介绍。

从下一章节开始，本书将进入《推荐系统：工业架构与核心算法》下册的内容。与上册侧重于系统架构与基

础算法不同，下册将更加聚焦于推荐系统中的优化问题、前沿研究以及新兴技术的发展，包括多目标优化、强化学习与推荐系统、因果推断、Agent、大语言模型 (LLM) 与推荐系统融合、生成式推荐等当前学术界与工业界广泛关注的研究方向。希望读者在完成上册学习之后，已经能够建立起对工业级推荐系统整体架构和核心模块的全局认知；而在下册中，我们将在这一基础之上，进一步深入探讨推荐系统背后的研究问题、算法演进以及未来的发展方向，从而帮助读者形成更加完整、系统且具有前瞻性的推荐系统知识体系。

18.9 参考文献

[1] Jon Louis Bentley. “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[2] Moses S Charikar. “Similarity estimation techniques from rounding algorithms”. In: *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 2002, pp. 380–388.

[3] Sanjoy Dasgupta and Yoav Freund. “Random projection trees and low dimensional manifolds”. In: *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 2008, pp. 537–546.

[4] Herve Jegou, Matthijs Douze, and Cordelia Schmid. “Product quantization for nearest neighbor search”. In: *IEEE transactions on pattern analysis and machine intelligence* 33.1 (2010), pp. 117–128.

[5] William B Johnson, Joram Lindenstrauss, et al. “Extensions of Lipschitz mappings into a Hilbert space”. In: *Contemporary mathematics* 26.189-206 (1984), p. 1.

[6] Yu A Malkov and Dmitry A Yashunin. “Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs”. In: *IEEE transactions on pattern analysis and machine intelligence* 42.4 (2018), pp. 824–836.

[7] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. “Detecting near-duplicates for web crawling”. In: *Proceedings of the 16th international conference on World Wide Web*. 2007, pp. 141–150.

[8] Stephen M Omohundro. “Five balltree construction algorithms”. In: (1989).

[9] 苏剑林. 让人惊叹的 *Johnson-Lindenstrauss* 引理：理论篇. Sept. 2021. URL: https://kexue.fm/archives/8679.