RAIRS：优化基于 IVF 的近似最近邻搜索中的冗余分配与列表布局

Zehai Yang yangzehai20z@ict.ac.cn SKLP，中国科学院计算技术研究所，中国科学院 中国科学院大学 北京，中国

Shimin Chen*chensm@ict.ac.cn SKLP，中国科学院计算技术研究所，中国科学院 中国科学院大学 北京，中国

arXiv:2601.07183v1 [cs.DB] 12 Jan 2026

摘要

IVF 是向量数据库中最广泛使用的近似近邻搜索（ANNS）方法之一。冗余分配的思路是将一个数据向量分配到多个倒排文件列表中，以降低 IVF 搜索中遗漏真实近邻的概率。然而，基于数据向量与列表质心之间的距离来选择第二个倒排文件列表的朴素策略表现不佳。以往的工作仅关注内积距离，而针对最常用的欧几里得空间，尚无经过优化的列表选择研究。此外，IVF 搜索可能在多个列表中访问同一个向量，从而导致冗余的距离计算并降低查询吞吐量。

本文提出 RAIRS，以应对上述两项挑战。针对列表选择这一挑战，我们为欧几里得空间提出了一种优化的 AIR 度量。AIR 不仅考虑距离，也考虑方向，从而支持这样的查询：查询向量更接近数据向量，却离首个选中列表的质心更远。针对冗余距离计算这一挑战，我们提出了 SEIL，这是一种优化的列表布局，利用共享单元减少 IVF 搜索中的重复距离计算。我们在具有代表性的真实世界数据集上进行实验，结果表明，RAIRS 优于现有的冗余分配方案，相较于性能最佳的 IVF 方法——带精化的 IVF-PQ 快速扫描，性能最高提升 1.33 倍。

CCS 概念

• **信息系统** → **数据库中的 Top-k 检索**；**数据访问方法**。

关键词

向量数据库；IVF 索引；近似最近邻搜索；冗余分配。

*Shimin Chen 为通讯作者。

作者联系信息：Zehai Yang，yangzehai20z@ict.ac.cn，SKLP，ACS，中国科学院计算技术研究所，中国科学院大学，北京，中国；Shimin Chen，chensm@ ict.ac.cn，SKLP，ACS，中国科学院计算技术研究所，中国科学院大学，北京，中国。

允许为个人或课堂使用而免费制作本作品全部或部分内容的数字副本或纸质副本，但不得为营利或商业利益制作或分发副本，且副本首页须附上本声明及完整引文。必须尊重本作品中第三方组件的版权。其他所有用途，请联系所有者/作者。*SIGMOD'26*，印度班加罗尔 © 2026年版权归所有者/作者所有。ACM
ISBN 978-x-xxxx-xxxx-x/YYYY/MM https://doi.org/10.1145/3786687

ACM 引用格式：
Zehai Yang 和 Shimin Chen。2026年。RAIRS：优化基于 IVF 的近似最近邻搜索中的冗余分配与列表布局。载于*SIGMOD'26* 论文集。 ACM，纽约，NY，美国，文章 73，共 16 页。https://doi.org/ 10.1145/3786687

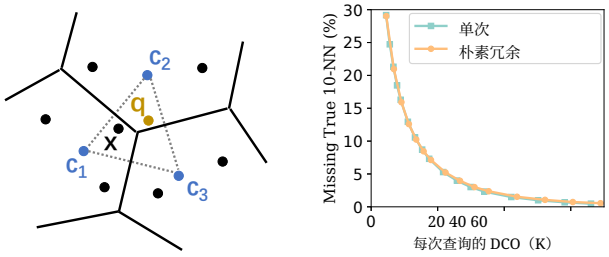
1 引言

向量搜索广泛应用于现实场景，包括推荐 [46], 数据挖掘 [14], 信息检索 [37], 以及近期的大语言模型（LLM）研究 [12, 61]。近似最近邻搜索（ANNS）是向量数据库中的关键操作。倒排文件索引（IVF）是应用最广泛的近似最近邻搜索方法之一。如图 1a 所示，在构建过程中，IVF 会计算 $nlist$ 个簇（也称为列表），并将每个数据向量分配到质心距离该向量最近的列表中。例如， x 被分配到 $list_1$，因为 x 比其他任何质心都更接近 c_1。给定查询 q后，IVF 会搜索列表质心，并选择距离 q最近的前$nprobe$个列表，然后遍历所选列表，计算其中所有向量的距离。在此例中，IVF 选择前 2 个列表（即 $list_2$和 $list_3$）。遗憾的是，它未能检索到 x、 q的真实最近邻。虽然增大 $nprobe$（例如从 2 增至 3）可能有所帮助，但 IVF 必须遍历更多列表，从而降低查询吞吐量。本文探讨了一种替代方案：冗余分配。其思路是将一个数据向量分配到多个 IVF 列表中。假设 x同时分配到 $list_1$和$list_2$。这样，遍历 $list_2$便可成功检索到 x，从而降低漏掉真实近邻的概率。

冗余分配面临的挑战。实现冗余分配理念主要面临两个挑战：

• 列表选择策略：朴素策略（NaïveRA）仅根据向量与列表质心之间的距离，为向量选择第二个列表。然而，实验结果表明，NaïveRA 几乎无法提升近似最近邻搜索性能。图 1b 比较了在 SIFT1M 数据集上进行 Top-10 近邻搜索时，NaïveRA 与单次分配的表现。X 轴表示每次查询的平均距离计算操作（DCO）次数，Y 轴表示漏掉的真实近邻所占百分比。可以看到，两条曲线几乎重合，这表明 NaïveRA 无法胜过基线IVF单次分配。

• 冗余距离计算：采用冗余分配时，IVF 搜索可能会在多个选定的列表中遇到同一个向量。例如，假设 x 被分配到 $list_1$ 和 $list_2$。如果查询时选中了这两个列表，IVF 就会访问 x 两次，导致对 x进行冗余距离计算。一种解决方法似乎是在计算距离之前，对所有选定列表中的向量 ID 去重



(a) 单次分配的问题 **(b) NaïveRA 效果不佳**

图 1：IVF 索引的冗余分配。

计算。然而，在先进的 IVF方法（例如 IVF-PQ 快速扫描）中，[3], 列表会将向量 ID 存储在打包块中，以便进行 SIMD 加速。为进行去重而解包这些块以获取单个 ID，代价非常高。

我们的解决方案：RAIRS。为应对这些挑战，我们提出了 RAIRS，它包含以下两种优化技术：

• *RAIR*（采用增强逆残差的冗余列表选择）。我们提出了一种优化的 AIR度量，用于欧几里得空间中的次级列表选择。将向量分配到质心最近的第一个列表后，RAIR 会选择使 AIR度量最小的第二个列表，即（$||r'||^2 + \lambda r^T r'$），其中 r, r' 分别是第一个和第二个列表的聚类残差，λ是一个常数参数。该度量不仅考虑距离（即第一项 $||r||^2$），还考虑方向（即第二项 $r^T r'$）。AIR 倾向于选择第二项为负的情况；它会选择这样一个第二列表：其质心相对于第一个已分配列表的质心，位于数据向量的逆方向，从而覆盖那些距离该向量更近、但距离第一个已分配列表的质心更远的查询。我们从形式上证明了 AIR 的有效性。此外，我们还研究了多列表分配，将 RAIR 扩展为选择三个或更多列表。

• *SEIL*（共享单元增强型 *IVF* 列表）。为减少冗余距离计算，我们提出了一种经过优化的列表布局 SEIL。我们用 $cell_{i,j}$ 表示同时分配给 $list_i$ 和 $list_j$ 的所有向量。根据真实世界数据分析，我们发现有些单元包含大量向量。在高级 IVF 方法中，例如 IVF-PQ 快速扫描，每 32 个向量项会被打包到一个块中，以实现 SIMD 加速。因此，我们将大于一个块的单元称为大型单元。我们发现，大量向量位于大型单元中。基于这些观察，我们设计了 SEIL，让两个列表（例如 $list_i$ 和 $list_j$）共享一个大型单元的块（例如 $cell_{i,j}$）。这类共享块可实现块级去重，从而避免对共享块重复进行距离计算。请注意，SEIL 不仅适用于 RAIR，也适用于任何冗余分配策略。

贡献。本文有三项贡献。首先，我们提出了 RAIR，这是一种面向欧几里得空间的新型冗余分配策略。我们从形式上证明了 AIR 分配度量的有效性。其次，我们提出了 SEIL，这是一种新型列表布局，可减少冗余分配导致的重复距离计算。最后，我们开展了广泛的实验研究，以评估 RAIRS 的优势。实验结果表明，RAIR 优于现有的冗余分配策略，SEIL 则能有效减少冗余距离计算。与带精化的 IVF-PQ 快速扫描（IVFPQfs）相比，RAIRS 在真实世界数据集上的查询吞吐量最高可提升 1.33 倍，同时保持相近的召回率。

概述。本文其余部分安排如下。第 2 节回顾相关背景，并讨论冗余分配所面临的挑战。第 3 节概述 RAIRS 解决方案，随后第 4 节和第 5 节分别提出 RAIR 和 SEIL。之后，第 6 节报告评估结果。第 7 节讨论相关工作。最后，第 8 节总结全文。

2 背景

本节回顾近似最近邻搜索、倒排文件和冗余分配的相关背景。

2.1 近似最近邻搜索

问题定义。给定一组 D 维向量 $X = \{x_1, x_2, \ldots\ldots, x_n\}$，其中 $x_i \in \mathbb{R}^D$ 对于 $i = 1, 2, \ldots\ldots, n$，以及一个查询向量 $q \in \mathbb{R}^D$，k近邻（kNN）问题旨在找出与 q 最接近的前 K 个向量。在当今的应用中，数据集规模通常极为庞大，包含数百万至数十亿个向量，而每个向量往往有数百甚至数千个维度。维度灾难 [29, 57] 使得在不进行穷举搜索的情况下无法找到精确的最近邻，而穷举搜索的代价可能高得令人难以承受。因此，业界和学术界的关注重点已转向近似最近邻搜索，以略微牺牲准确性为代价，大幅提升处理速度和可扩展性。

距离度量。最常用的距离度量是欧几里得距离：$dist(q, x) = \sqrt{\sum_{i=1}^{D}(q^{(i)} - x^{(i)})^2}$，其中 $x, q \in \mathbb{R}^D$。其他常见的距离度量还包括内积、余弦相似度等。本文提出的冗余选择方法 RAIR 针对欧几里得距离，而优化后的列表布局 SEIL 则适用于所有距离度量。

2.2 倒排文件索引（Inverted File Index）

IVF 是向量数据库中应用最广泛的近似最近邻搜索方法之一。接下来，我们将介绍实践中性能最佳的 IVF 变体——带精化的 IVF-PQ 快速扫描 [3, 5]，并将其用作本文的基线。

• 乘积量化：PQ [31] 是一种广泛采用的量化方法，可加速距离计算。它将向量维度划分为若干组（例如，每组包含 2 个维度），并据此将每个数据向量划分为多个子向量。随后，对于每个维度组，PQ 将子向量划分为（例如，使用 K均值得到的）若干簇（例如 16 个）。它将一个向量编码为其子向量对应的簇 ID（也称码字）。IVF-PQ 将 PQ 编码与向量 ID 一同存储在 IVF 列表中。给定一个查询，IVF-PQ 会构建查找表（Look-Up Tables, LUT），其中包含每个维度组内查询子向量与所有簇质心之间的平方距离。为了估算$dist(q, x)$，IVF-PQ 会针对 x 的子向量中的每个码字查找相应的查找表，并将平方距离累加起来。

• 精化：由于估计距离不够精确，通常会将 PQ 等量化方法与精化结合，以提升搜索质量。其思路是检索更多

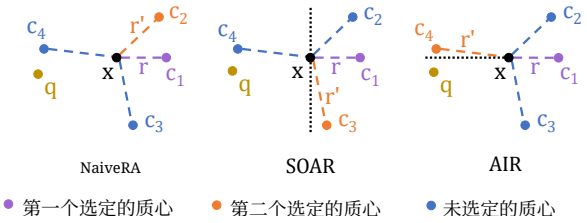


图 2：不同的冗余分配策略。

个向量（例如，每个查询取 10×K 个），这些向量来自经过量化增强的 IVF 索引；然后计算精确距离，并对检索到的向量重新排序，以获得排名前 K 的结果。

• *PQ* 快速扫描：PQ 快速扫描是多种现有技术之一，这些技术利用 SIMD 加速进行距离计算。在索引构建过程中，PQ 快速扫描会将倒排文件列表中的数据项（即 PQ 编码和向量 ID）组织成紧凑的固定大小（例如，每块 32 项）的向量块，以便于 SIMD 访问。随后，在查询处理过程中，它会将查找表加载到 SIMD 寄存器中，并使用 SIMD 指令一次计算一个数据块中各项的距离。

2.3 冗余分配

冗余分配会将每个向量项分配到多个倒排文件列表，而非单个列表，从而降低遗漏真实前 K 个最近邻的可能性；这些最近邻原本可能只被分配到距离查询向量较远的列表中。

NaïveRA（朴素冗余分配）仅依据距离进行列表选择。SPANN [13] 使用 NaïveRA 将部分向量放入 SSD 页中的多个列表。相比之下，我们关注的是内存驻留环境。

SOAR [50] 是一种针对内积距离的冗余分配策略。给定第一个列表，它会选择使 $||r'||^2 + \lambda(\ r^T r, ||r||)^2$, 最小的第二个列表，其中 $r, r,$ 分别是第一个和第二个列表的聚类残差。第二项非负，并在 r 与 $r,$ 接近正交时达到最小值。

图 2 展示了 NaïveRA 和 SOAR。x 是一个向量。c_1、c_2、c_3 和 c_4 是四个列表质心。由于 c_1 是最近的质心，因此 x 首先被分配到 $list_1$。然而，对于查询 q，x 是 q 的真实最近邻，但 c_1 距离 q 很远。因此，对 q 进行 IVF 搜索时，很可能会漏掉 x。我们希望对 x 进行冗余分配。如图 2 所示，NaïveRA 只是选择第二近的质心 c_2。SOAR 则选择 c_3，其残差向量 $r' = x - c_3$ 接近于与主残差向量 $r = x - c_1$ 正交。遗憾的是，c_2 和 c_3 都不是 q 的理想选择。

本文提出了一种针对欧几里得空间优化的 AIR 策略。为了支持这类查询 q：它们与向量的距离更近，但距离第一个分配列表的质心更远，AIR
会选取第二个列表，使其残差方向与主要残差相反。如图2所示，c_4 被选中，能够很好地支持 q。

3 RAIRS 概述

我们概述 RAIRS 的数据结构、提出的两项优化以及索引操作。

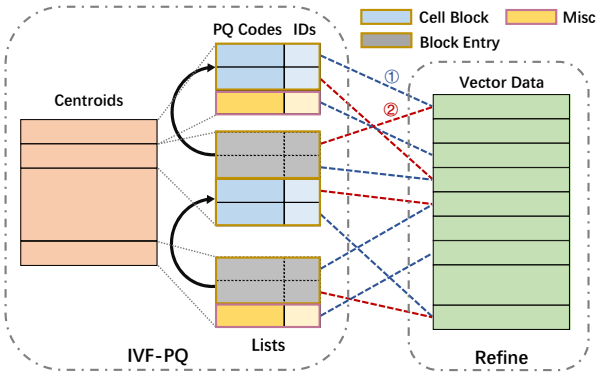


图3：RAIRS索引概览。

数据结构。 RAIRS 继承了带精炼功能的 IVF-PQ 数据结构。如图3所示，它包含三个主要分量：1）质心，2）倒排列表，3）向量数据。前两个分量构成 IVF-PQ 模块，而精炼模块保留原始向量数据。每个向量最多分配到两个列表，如蓝色和红色虚线所示。倒排列表存储 PQ 编码和向量 ID，所占内存空间远小于原始向量。在查询时，近似最近邻搜索首先通过 IVF-PQ 检索一组候选向量。随后，它访问精炼模块，为候选项计算精确距离，并对候选项重新排序，以获得最终的 top-K 个结果。

RAIR。 给定查询 q，倒排文件（IVF）会遍历质心距离 q最近的 $nprobe$ 个列表。但是，如果 x 不在距离 q最近的 $nprobe$ 个列表中，IVF 就会漏掉 q的前K 个最近邻 x。为了解决这个问题，我们提出了一种 AIR（放大逆残差）度量，用于优化欧几里得空间中的冗余分配。RAIR 使用 AIR 度量，将每个向量分配到第二个倒排文件列表。如此一来，在相同的 $nprobe$下，找到真正的前K 个结果的机会便会提高。从另一个角度来看，它可以在获得相近搜索准确率的同时减少遍历列表的数量，从而提升查询吞吐量。此外，我们还考虑第一个和第二个选定列表相同的情况，并将 RAIR 推广到多重分配。（参见第4节）

SEIL。 我们使用"x 位于 $cell_{i,\ j}$"这一表述，表示向量 x 被分配到 $list_i$ 和 $list_j$，并进行了冗余分配。我们观察到，部分单元包含大量向量。各单元中的向量数量存在明显偏斜。这一有趣发现促使我们优化大型单元的列表布局，以减少冗余距离计算。

在基线中，每个列表都被划分为打包的 32项块，以便通过 PQ 快速扫描进行 SIMD 计算。一个大型 $cell_{i,\ j}$会同时存储在 $list_i$和 $list_j$中。如果在同一个查询中遍历这两个列表，则会对 $cell_{i,\ j}$执行两次距离计算，这是浪费的。为了解决这一问题，我们提出了 SEIL，仅存储一次 $cell_{i,\ j}$的共享块。在图3中，共享单元块以蓝色表示。灰色的块入口指向共享单元块。对于一个查询，SEIL 仅对共享块执行一次距离计算，从而减少冗余距离计算。（参见第 5 节）

索引构建。 算法1展示了将一批向量添加到 RAIRS索引中的过程。对于每个向量，算法都会调用*RairAssign* 将该向量分配到两个列表中（第 4 行），并

算法1 向 RAIRS索引添加一个向量批次。

输入： 索引，向量，vec_ids

```
1: 函数 AddVectors(index, vecs, vec_ids)
2:     分配= []；编码 = [];
3:     对于 (i = 0; i < vecs.len; i ++)执行
4:         (列表 ID1, 列表 ID2) = RairAssign(索引, vecs[i]);
5:         assignments.append( {列表 ID1, 列表 ID2, vec_ids[i]} );
6:         codes.append( PQ编码(vecs[i], index.code_book) );
7:         index.vec_data.append(vecs[i]);
8:     SeilInsert(索引, 分配, codes, vec_ids);
9:     索引.ntotal += vecs.len;
```

算法 2 使用 RAIRS 索引的近似最近邻搜索。

输入： 索引、查询、K、nprobe
输出： 结果

```
1: 函数 Rairs搜索(index, queries, K, nprobe)
2:     bigK = K * K_因子;
3:     对于 (i = 0; i < queries.len; i ++)执行
4:         查找表 = 计算查找表(查询[i], 索引.code_book);
5:         所选_列表 = FindNearestLists(index, queries[i], nprobe);
6:         候选项 = SeilSearch(index, 查找表, selected_列表, bigK);
7:         results[i] = Refine(index.vec_data, queries[i],候选项, K);
8:     返回 results;
```

计算其 PQ 编码（第 6 行）。同时，原始向量也会追加到向量数据中，以便精炼模块进行精确的距离计算（第 7 行）。最后，算法调用*SeilInsert*，在进行 SEIL 布局优化的同时，将向量项插入倒排列表（第 8 行）。*RairAssign* 和*SeilInsert* 将在第 4.2 节和第 5.3 节中分别详细介绍。

ANNS 查询处理。 算法 2 列出了针对一批查询寻找前 K 个最近邻的 ANNS 过程。首先，它将从倒排文件列表中检索的候选项数量（*bigK*）设为 K 乘以预定义的 *K_FACTOR*（例如 10）（第 2 行）。对于每个查询向量，该算法会构建 PQ 距离查找表（*LUT*）（第 4 行），并确定质心距离查询向量最近的 *nprobe* 个列表（第 5 行）。然后，它调用 *SeilSearch* 遍历 SEIL 结构中的每个相关列表，根据 *LUT* 和 PQ 编码计算近似距离，并检索一组 *bigK* 个候选项（第 6 行）。最后，精炼模块根据精确距离计算得出前 K 个结果（第 7 行）。第 5.3 节详细介绍了 *SeilSearch* 如何高效地获取所需候选项，同时减少冗余距离计算。

适用性。 本文假设向量数据能够存放在主存中。我们以带精化的 IVF-PQ 快速扫描作为基线，以展示我们方法的有效性提出的 RAIR 和 SEIL 优化。

请注意，RAIR 可应用于任何基于 IVF 的索引。冗余分配策略与索引所采用的量化方法、存储介质和硬件优化相互正交。此外，尽管 SEIL 旨在支持 PQ 快速扫描中的打包块，但利用共享单元来减少冗余距离计算的思想具有普适性。例如，对于磁盘驻留 IVF-flat 索引，其将 IVF 列表存储在磁盘上，并将质心存储在内存中；我们可以通过用向量的共享单元替换磁盘上的列表，并将共享单元地址与列表质心一同记录在内存中，来应用 SEIL 的思想。

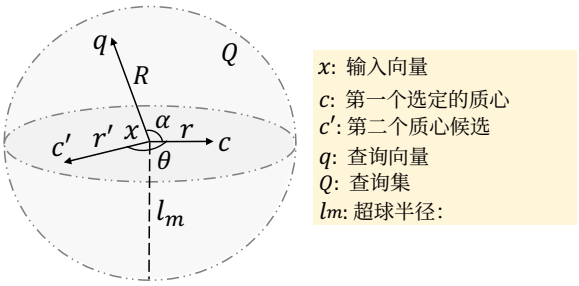


图 4：向量的几何关系。

4 RedundantListSelection withAIR

我们在第 4.1 节提出了 AIR（增强逆残差），将其作为欧几里得空间中的一种优化列表选择度量。随后，我们在第 4.2 节介绍并分析 RAIR 算法，以将一个向量分配到两个列表。最后，我们在第 4.3 节研究将 RAIR 推广到把一个向量分配到多个列表的情况。

4.1 AmplifiedInverse Residual

设 x 为要插入倒排文件列表的数据向量， c 为距离 x 最近的质心。我们考虑与 x 的距离不超过最大距离 l_m 的查询。设 $Q = \{q: ||q-x|| \le l_m\}$ 为以 x 为中心、半径为 l_m 的超球体内所有查询的集合。假设 $q \in Q$ 是在 Q 中均匀分布的随机查询向量。图 4 展示了这些向量的几何关系。

由于 c 是距离 x 最近的质心，与质心 c 关联的列表是最先被选中用于分配 x 的列表。在大多数情况下，用 c 表示 x 已足够。然而，对于某些查询 q， c 可能并非表示 x 的理想选择。也就是说， c 位于 q 最近的 *nprobe* 个列表之外，因此， x 虽然是 q 的真正前 K 近邻，却不会出现在采用单次分配检索到的结果中。在这种情况下，尽管 q 接近 x，但 q 离 c 太远，以至于有 *nprobe* 个其他质心比 c 更接近 q。

我们希望为 x 选择第二个列表，以满足这类查询的需求：它们无法从质心为 c的第一个选定列表中获得足够的收益。如图 4 所示，令 $\alpha = \angle qxc$。 α量化了对于查询 q而言，将 c选作x的第一个质心有多不理想。这是因为在三角形 $\triangle qxc$中，较大的 α表示边 qc较长，而这表示 $dist(q,\ c) = ||q-c||$。也就是说， α越大，q离 c越远， c出现在最近的 q个列表中的可能性就越低。

基于这一直觉，我们构造了以下用于选择第二个列表的损失函数。令 c' 表示第二个选定列表的质心。我们确保第二个质心 c' 能够补偿 c 中所有查询的 Q:

$$L(c', c, Q) = E_{q \in Q}[ReLU(-\cos \angle qxc) \cdot (||q-c'||^2 - ||q-x||^2)]$$

在损失函数中，第二个因子 $||q-c'||^2 - ||q-x||^2$ 很容易理解。它表示，相较于 q 到 x，更倾向于减小 q 到 c' 的欧几里得距离平方。第一个因子 $ReLU(-\cos\ \alpha)$ 是一个加权项，表示第二个质心 c' 对 q 的重要程度。当 $\alpha \le \frac{\pi}{2}$ 时， x 很可能会在由 c 表示的列表中被 q 访问。在这种情况下， $-\cos\alpha \le 0$ 和 $ReLU$ 返回 0。因此，第二个因子的贡献会被忽略，这意味着第二个列表并不重要。另一方面，如果 α 大于 $\frac{\pi}{2}$，则 $ReLU(-\cos\alpha) > 0$。

表 1：冗余分配策略比较。

策略	NaïveRA	SOAR	AIR
公式	$\|\|r'\|\|^2$	$\|\|r'\|\|^2 + \lambda(\frac{r^T r'}{\|\|r\|\|})^2$	$\|\|r'\|\|^2 + \lambda r^T r'$
几何解释	第二近邻邻居	优先选择第二残差与……正交第一残差	优先选择第二残差与……相反第一残差

注：r（r'）是第一个（第二个）选定列表的聚类残差。

q 接近 x，但远离 c。我们会提高这类查询的权重，使其在分配过程中获得更高优先级。α 越大，权重越高。然后，我们从所有列表中选择 $L(c',\ c,\ Q)$ 最小的第二个列表。

我们证明以下定理，以简化损失函数的计算。完整证明见附录。

定理 4.1. 对于一组 Q 在以 x 为中心、半径为 l_m的超球体内均匀分布的查询，

$$L(c', c, Q) \propto ||r'||^2 + \lambda r^T r'$$

其中 $r = c - x$、$r' = c' - x$和 $\lambda > 0$ 是常数因子。

AIR. 根据定理 4.1，我们将 $||r'||^2 + \lambda r^T r$，设为选择第二个列表的度量。从公式中可以看出，该度量并非只依赖于最小化 $||r'||$，即第二个选定质心 c'到 x的距离。此外，还增加了惩罚项 $\lambda r^T r$，。当 $||r||$固定时，若 r，更接近 r的逆向量（即 $-r$），则会使第二项为负，从而降低损失函数。这表明，该度量倾向于使第二残差 r'与第一残差 r处于相反的方向。因此，我们将此度量称为 AIR（放大逆残差），以体现其对逆残差的偏好。

— - —

有趣的是，如果设置 $\lambda = 0$，AIR 就会退化为 NaïveRA，后者会选择第二近列表作为第二选择。在我们的实验中，默认设置为 $\lambda = 0.5$。我们还在第 6.3 节深入研究了 λ 对近似最近邻搜索性能的影响。

冗余分配策略比较。表 1 对比了 NaïveRA、SOAR 和 AIR。首先，NaïveRA 旨在最小化第二个选定列表的列表质心距离。相比之下，SOAR 和 AIR 不仅考虑距离，还考虑第一个与第二个选定列表之间的关系，这一点可从其公式的第二项看出。其次，SOAR 和 AIR 存在显著差异。SOAR 的第二项与第二残差 r' 在第一残差 r 上的投影平方成正比。因此，SOAR 倾向于使 r，与 r 正交，从而使第二项接近 0。相比之下，AIR 的第二项计算两个残差的点积，该值可以为负。AIR 倾向于使 r' 处于 r 的逆方向。我们在第 6 节通过实验比较了 NaïveRA、SOAR 和 AIR。

4.2 RAIR 算法

算法 3 展示了 *RairAssign* 过程。给定一个数据向量v，算法首先获取 N_CANDS个质心距离 v最近的列表（第 2 行）。随后，按列表质心与 v之间的距离升序排列 $cand_lists$。接着，计算所有候选列表的 AIR 损失函数（第 3–8 行）。之后，算法选择主列表，将 v分配为 $cand_lists[0]$；该列表的质心距离 v最近（第 9 行）。算法确定

算法 3使用 RAIR 将向量分配到列表中。

输入： 索引，v
输出： 列表 ID1，列表 ID2

```
1: 函数RairAssign(index, v)
2:     cand_列表 = FindNearestLists(索引, v, N_CANDS);
3:     质心0 = 索引.质心[候选_列表[0]];
4:     残差0 =质心0 − v;
5:     对于(i = 0; i < N_CANDS; i ++) 执行
6:         质心 = 索引.质心[候选_列表[i]];
7:         残差 =质心 − v;
8:         损失[i] = L2平方(残差) + λ· 内积(残差0, 残差);
9:     列表 ID1 = 候选_列表[0];
10:    起始值 = （是否为_严格模式）？ 1： 0;
11:    列表 ID2 = 候选_列表[ argmin(损失[起始值 .. N_候选-1]) ];
12:    返回 列表 ID1 < 列表 ID2 ? (列表 ID1, 列表 ID2) : (列表 ID2, 列表 ID1);
```

次级分配时，选择使计算得到的 AIR 损失函数最小的列表（第 11 行）。

RAIR 和严格 RAIR（SRAIR）。当 AIR 损失函数在 v的主列表上保持最小值时，即对于任意列表，

$$||r'||^2 + \lambda r^T r' \geq (1 + \lambda)||r||^2,$$

将 v 分配到任何次级列表几乎没有益处。因此，这些向量仅存储在其首选列表中。该策略不仅通过限制不必要的冗余来减少空间开销，还能减少查询时对向量的不必要访问，从而有望提升整体查询性能。我们将此策略称为 RAIR。

除了 RAIR，我们还提供了一个严格版本的 RAIR，称为 SR AIR。SRAIR 严格地为一个向量分配两个列表。也就是说，它排除第一个选定的列表，并应用 AIR 从其余列表中选择第二个列表。

该算法使用 is_strict标志来支持 RAIR 和 SRAIR（第 10 行）。当 is_strict为 false 时，$start$=0，并从所有 N_CANDS列表中选择第二个列表（第 10–11 行）。当 is_strict为 true 时，$start$=1，并且不考虑 $cand_lists[0]$（即第一个选定的列表）（第 10–11 行）。

通过限制候选列表降低计算成本。在实践中，我们不会针对每个向量计算所有 $nlist$列表的损失函数。相反，我们只评估排名前 N_CANDS的最近列表，如算法 3 所示。请注意，这对于大型数据集非常重要，因为此时 $nlist$很大。*FindNearestLists*可以执行近似最近邻搜索，而不是穷举搜索，从而降低每个向量最坏情况下的 $O(nlist \cdot D)$ 成本。

我们发现，较小的 N_CANDS（例如 10）通常足以保证冗余分配的质量。例如，在 SIFT1M [30] 数据集中，对于超过 99.95% 的向量，当 $\lambda = 0.5$ and . 时，最小损失函数是在前

$$\frac{10}{1024} \quad \frac{N_CANDS}{nlist} = \text{个最近列表中取得} \quad 6\,3$$

的。我们将在第 . . 节深入研究 的设置。

成本分析。首先，选择第一个列表包括调用 *Find- NearestLists*（第 2 行），并在算法 3 中设置 $listID1$（第 9 行）。根据实现方式，如果 *FindNearestLists*执行穷举搜索，其成本为 $O(nlist \cdot D)$；如果 *FindNearestLists*执行近似最近邻搜索，则成本可降至 $O(sublinear(nlist) \cdot D)$。（例如，如果 *FindNearestLists*执行基于倒排文件的近似最近邻搜索，其复杂度可为 $O(\sqrt{nlist} \cdot D)$。）

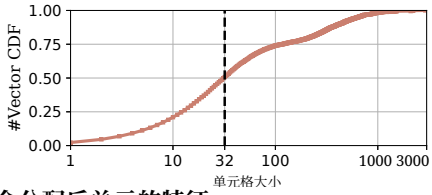


图5：冗余分配后单元的特征。

其次，算法的其余步骤使用 D 维向量计算，从 N_CANDS 个候选向量中选出第二个列表。因此，成本为 $O(N_CANDS \cdot D)$。

总体而言，时间复杂度可表示为 $O((nlist + N_CANDS)D)$ 或 $O((sublinear(nlist) + N_CANDS)D)$，具体取决于 *FindNearestLists* 的实现。

请注意，N_CANDS通常比 $nlist$小好几个数量级。选择第二个列表所需的额外开销通常远小于选择第一个列表的开销。因此，冗余分配中的列表选择开销通常接近基线单次分配的开销。

4.3 推广到多次分配

在上述方法中，RAIR 执行双分配，将每个向量分配到最多两个倒排文件列表。本小节将 RAIR 推广到 m 分配，其中 $m \geq 3$。

给定已选定的 $(m-1)$ 个列表，我们通过考量相对于所有先前已选质心的损失来选择第 m个质心：

$$L_m(c', c_1, ..., c_{m-1}, Q) = \underset{1 \leq i \leq m-1}{aggr} L(c', c_i, Q) \propto ||r'||^2 + \lambda \underset{i}{aggr}\ r_i^T r'$$

对于 $aggr$，我们在第 6.3 节评估了三个函数（即 max、min 和 avg）。结果表明，max 表现最佳。

请注意，将向量分配到两个以上的列表并不一定能提升近似最近邻搜索性能。将每个向量分散到更多列表中，可以减少查询所需遍历的列表数量（$nprobe$）。然而，IVF 列表平均大小会增大，从而导致更多距离计算操作。我们的实验表明，双分配所需的距离计算次数最少。

5 共享单元增强型 IVF 列表

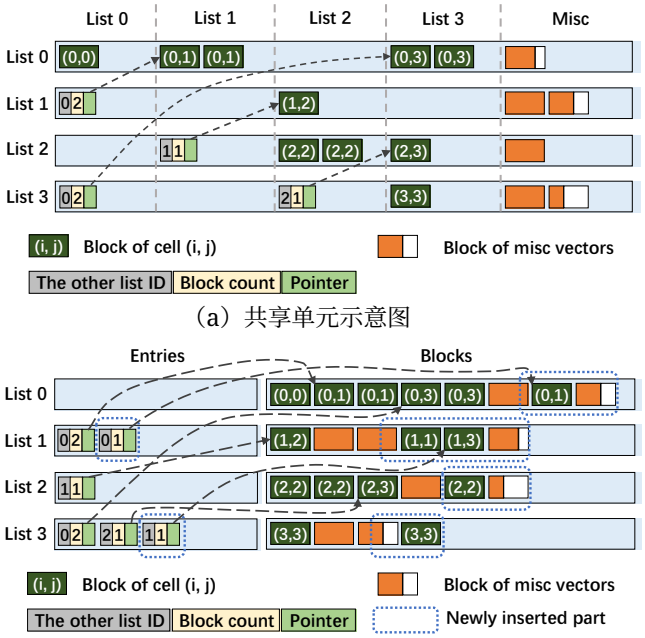
我们在第 5.1 节讨论基线列表布局在支持冗余分配方面的问题。接着，我们在第 5.2 节提出 SEIL 布局优化。最后，我们在第 5.3 节介绍 SEIL 增强索引的算法。

5.1 基线列表布局的挑战

考虑向量 x 被分配到 $list_i$ 和 $list_j$ 的情况。在基线列表布局中，向量项（包括 x 的 PQ 编码及其向量 ID）会同时存储在 $list_i$ 和 $list_j$ 中。PQ 快速扫描还会将每 32 个向量项打包成一个块，以便在每个列表中实现 SIMD 加速。这种基线列表布局存在两个问题：1）如果查询选中的倒排文件列表中同时包含

$list_i$ 和 $list_j$，则查询时会产生冗余距离计算；2）存储两份向量项会增加空间开销。

第一个问题会降低查询吞吐量。一种简单直接的解决方案是收集 $nprobe$ 所选列表中的所有向量 ID，去重后再进行距离计算。另一种



（a）共享单元示意图

(b) 两批插入后的列表结构

图6：SEIL 列表布局。

解决方案是建立一个哈希表，记录已计算距离的向量 ID，然后在计算过程中查询哈希表，以避免任何冗余计算。然而，这两种解决方案都需要在距离计算之前获取向量 ID。不幸的是，这需要解包打包块，从而打断 SIMD 计算步骤，大幅削弱 PQ 快速扫描的优势。此外，这两种解决方案都对 $nprobe$ 列表中的所有向量项采用基于排序或哈希的算法，这可能会带来不容忽视的额外时间和空间开销。

第二个问题通过在列表中复制向量项而非 D 维向量来缓解。此外，如果第一个和第二个分配列表相同，RAIR 就不会重复存储向量项。不过，进一步降低冗余分配带来的空间开销仍是值得考虑的。

5.2 SEIL 布局

单元的特征。我们研究图5中单元的特征。请注意，$cell_{i,\ j}$包含所有同时分配给 $list_i$ 和 $list_j$ 的向量。由于 $cell_{i,\ j}$ 和 $cell_{j,\ i}$ 基本相同，我们确保 $i \leq j$ 对于 $cell_{i,\ j}$成立。对于在 RAIR 中仅分配到一个列表 i 的向量，我们将其单元设为 $cell_{i,\ i}$。我们统计冗余分配后每个单元中的向量数量。x 轴以对数刻度显示单元大小（即一个单元中的向量数量）。y 轴展示 SIFT1M 数据集中向量在各单元间分布的累积分布函数（CDF）。1.0 对应所有单元大小之和。

在图5中，我们在单元格大小为 $= 32$ 处画了一条虚线。由于 PQFast Scan 中一个块包含 32 个向量项，因此我们将单元格大小为 ≥ 32 的单元视为大型单元。在图中，虚线右侧是大型单元。从图中我们观察到：1）大型单元包含了约 50% 的所有向量；2）存在一些非常大的单元，其中包含数百到数千个向量。

观察到向量高度集中在大型单元中，这促使我们利用共享单元。我们的想法是尽可能在 $list_i$ 和 $list_j$ 中共享大型 $cell_{i,j}$ 的向量，以减少查询时的冗余距离计算，并降低冗余分配带来的空间开销。

SEIL 列表结构。SEIL 列表结构如图 6 所示。图 6(a) 展示了共享单元的概念。$cell_{i,j}$ 绘制在第 i 行、第 j 列。单元中的向量被分组成 32项块。假设一个单元包含 $nitems$ 个向量。那么，我们会生成 ($nitems$/32) 个块。剩余的 ($nitems$%32) 个向量会追加到杂项区域中的块里。$cell_{i,j}$ 的块在内存中只存储一次。它们由 $list_i$ 和 $list_j$ 共享。例如，$cell_{0,1}$ 的块由 $list_0$ 和 $list_1$ 共享。这些块在物理上存储于 $list_0$。$list_1$ 包含一个（另一列表 ID、块计数、指针）条目，用于引用 $list_0$ 中的共享块。

图 6(b) 展示了物理列表结构。每个列表都包含一个引用条目数组和一个 32项块数组。$cell_{i,j}$ ($i < j$) 的共享块在物理上存储于 $list_i$，而 $list_j$ 存储指向物理块的引用条目。图中还展示了经过两个插入批次后的结构。第二批次的条目和块以虚线圆角矩形突出显示。它们被追加到条目数组和块数组的末尾。

我们想指出几项设计考量。首先，一个引用条目可以指向同一单元的多个块。对于一个插入批次，SEIL 会将同一单元的多个块连续存储。引用条目通过另一列表 ID 和指针字段指向第一个块。块计数是该单元中连续块的数量。其次，对于新的一批向量，可以生成一个引用同一另一列表的新条目。例如，对于第二批次，$cell_{0,1}$ 会看到一个新块。随后，新块存储在 $list_0$中，并将一个指向 $list_0$新块的引用条目追加到 $list_1$。现在，$list_1$ 包含两个都指向 $list_0$的引用条目。第三，一个列表最后的杂项块可能包含少于 32 个向量项，如图中的半满橙色矩形所示。其他块均为满块。对于新的批次，我们会先填满前一批次的最后一个杂项块，再生成新的杂项块。最后，对于存储在杂项块中的向量项，我们会将其分配到的另一列表 ID 嵌入向量 ID 未使用的高位比特中。假设 x是 $cell_{i,j}$中剩余的 ($nitems$%32) 个向量之一。那么，在将 x存储到 $list_i$的（$list_j$的）杂项块时，我们会嵌入 $j(i)$。这简化了杂项向量的去重。

SEIL 优化去重。对于共享单元中的块，SEIL 通过单元级去重来减少冗余距离计算。其基本思路是判断某个共享单元是否已在同一查询中处理过；如果该单元已被访问，则跳过该共享单元中所有块的 SIMD 距离计算。

为实现这一点，我们区分物理块和引用条目。对于共享单元中物理存储的块，我们始终执行距离计算。对于引用条目，我们需要判断同一查询中是否已访问过其对应的块。这通过一个 $listVisited$哈希表实现，该表用于跟踪查询中已访问的列表。使用引用条目的另一列表 ID 对 $listVisited$进行探测。如果该 ID 已存在，则相应的块已处理过，因此跳过该引用条目。否则，将对该引用条目所代表的块执行距离计算。

对于杂项区域中的向量，由于 PQ FastScan 的打包块，我们无法避免距离计算。不过，距离计算后仍有必要对向量去重，以免同一向量在返回结果中出现两次。我们不必详细记录所有已访问的杂项向量，而是利用上述 $listVisited$哈希表来简化去重。具体而言，对于每个杂项向量x，我们检查嵌入在 x 的向量 ID 中的另一列表 ID 是否已在 $listVisited$中被访问；如果检查结果为真，则立即跳过该向量。

5.3 SEIL 算法

构建 SEIL 优化列表。算法 4 展示了将一批向量插入 SEIL 优化列表的过程。该算法按升序对分配项进行排序，使同一单元中的项在 $assigns$ 数组中连续排列（第 8 行）。随后，算法通过两个 for 循环对 $assigns$ 数组扫描两次。

第一个 for 循环计算每个列表的共享块数量和杂项区域中的项目数量（第 9–15 行）。每次循环迭代都会处理同一单元中的项目。它获取该单元（第 10 行），统计单元中的项目数量（第 11 行），计算块数量和剩余项目数（第 12 行），并更新相关统计信息（第 13–15 行）。之后，这些信息将用于为列表分配空间（第 16 行）。

第二个 for 循环将项目填入列表（第 17–32 行）。它以与第一个 for 循环相同的方式获取 $cell$、$nitems$、$nblocks$和 $nmisc$（第 18–20 行），然后将共享块（第 23–25 行）和杂项项目（第 26–27 行）追加到第一个列表中。如果第二个列表与第一个列表不同，算法还会将杂项项目追加到第二个列表中（第 31–32 行），并创建一个指向第一个列表共享块的引用条目（第 30 行）。同一块中的向量按 PQ Fast Scan 格式排列其 PQ 编码。

搜索 SEIL 优化列表。 算法 5 使用两个结构来辅助搜索：$rqueue$用于维护前 $bigK$ 个向量， $listVisited$用于跟踪已访问列表以便去重。算法先初始化这两个结构（第 2–3 行），随后进入循环，搜索每个选定的列表（第 4–18 行）。

对于每个列表，算法会处理引用条目（第 6–10 行）、存储在当前列表中的共享块（第 11–13 行），以及杂项区域中的块（第 14–17 行）。对于引用条目，算法使用单元级去重来检查是否可以跳过某个条目所指向的所有块（第 7 行）。对于物理存储的共享块，由于会在另一个列表中检查相应的引用条目，因此可以保证不会发生重复。对于杂项区域中的向量，则会在距离计算后使用 $listVisited$ 去重（第 16 行）。

对每个块都会调用 PQ 快速扫描，以使用 SIMD 指令高效计算距离（第 9、12、15 行）。在每次循环迭代结束时，会将当前列表添加到 $listVisited$（第 18 行）。最后，算法返回前 $bigK$ 个候选项（第 19 行）。

实现：优化查询批次的缓存。近似最近邻搜索可用于一批查询，这在各种

算法 4将一批向量插入 SEIL 优化列表。

```
输入：索引、分配项、编码、向量_ID
1: 函数获取单元格(分配项)
2:     返回 (分配项.列表 ID1, 分配项.列表 ID2);
3: 函数 统计同单元格项目数(assigns, i, cell)
4:     对于 (j = i+1; j < assigns.len; j++)执行
5:         如果 (获取单元格(assigns[j]) != cell) 则 break;
6:     返回j − i;
7: 函数 SeilInsert(索引, assigns, codes, vec_ids)
8:     按 {列表 ID1, 列表 ID2, vec_id} 的升序对 assigns 排序;
9:     对于(i = 0; i <分配项.len; i +=nitems)执行
10:        单元 = 获取单元格(assigns[i]);
11:        nitems = 统计同单元格项目数(assigns, i, cell);
12:        块数 = nitems / BLK_SZ; 杂项数 = nitems % BLK_SZ;
13:        列表_nb[cell.listID1] += nblocks; 列表_nm[cell.listID1] += nmisc;
14:        如果 (单元.列表 ID1 != 单元.列表 ID2)则
15:            列表_nm[单元.列表 ID2] += 杂项数;
16:    根据列表_nb 和列表_nm 分配空间;
17:    对于(i = 0; i <分配项.len; i +=项目数)执行
18:        单元 = 获取单元格(分配项[i]);
19:        项目数 = 统计同单元格项目数(分配项, i, 单元);
20:        块数 = 条目数 / BLK_SZ; 杂项数 = 条目数 % BLK_SZ;
21:        列表1 = 索引.lists[单元.listID1];
22:        bptr =列表1.getNextBlockPointerInSharedCell();
23:        for (b = 0; b < 块数; b++)do
24:            begin = i + b*BLK_SZ; end = begin + BLK_SZ − 1;
25:            list1.appendBlockInSharedCell(分配项, begin, end);
26:        对于(j = i + 块数*BLK_SZ; j < i+nitems; j++) 执行
27:            list1.appendItemInMiscArea(分配项, j);
28:        如果(cell.列表 ID1 != cell.列表 ID2)那么
29:            列表2 = 索引.列表[单元.列表 ID2];
30:            list2.追加引用条目(cell.列表 ID1, 块数, bptr);
31:            对于(j = i + 块数*BLK_SZ; j < i+nitems; j++) 执行
32:                list2.appendItemInMiscArea(分配项, j);
```

算法 5搜索 SEIL 优化列表。

```
输入：索引、查找表、选定的列表、bigK
输出：候选项
1: 函数SeilSearch(索引, 查找表, 选定的列表, bigK)
2:     rqueue.init(bigK);
3:     listVisited.init();
4:     对于(each listID in selected_lists) 执行
5:         列表 = 索引.lists[listID];
       /* 处理引用条目 */
6:         对于(each (otherLID, nblocks, bptr) in list.ref_条目) 执行
7:             如果 (! listVisited.exist(otherLID))则
8:                 对于(b = 0; b < 块数; b ++) 执行
9:                     结果 = PQFastScan(查找表, bptr[b]);
10:                    rqueue.update(results);
       /* 处理共享块，不进行重复检查 */
11:        对于(列表.共享_单元_块)执行
12:            结果 = PQFastScan(查找表, 块);
13:            rqueue.update(结果);
       /* 处理杂项区域中的项目 */
14:        对于(列表.杂项_块中的每个块)执行
15:            结果 = PQFastScan(查找表, 块);
16:            结果 = RemoveVectorIfVisited(listVisited, 结果);
17:            rqueue.update(results);
18:        listVisited.add(listID);
19:    返回rqueue.output(bigK);
```

近似最近邻搜索基准测试 [5, 69] 中很常见。将每个（查询、选中的列表）视为一个计算任务。一种实现方式是按查询对任务分组，并在处理下一个查询之前，先处理完同一查询的所有任务。然而，这种方式可能并非最优。在许多情况下，同一查询批次中的多个查询都会访问同一个列表。由于单个查询访问的数据通常远大于 CPU 缓存，因此这种实现会因多个查询反复访问同一个列表而产生大量 CPU 缓存未命中。

为了解决这个问题，我们的实现采用了一种优化技术来提升 CPU 缓存性能，Milvus [52] 和 Faiss [18]中均有此技术。其思路是按列表对任务分组，并连续处理针对同一列表的所有查询。这样，除了第一个搜索该列表的查询之外，该列表都会一直保留在 CPU 缓存中。为了简洁起见，我们省略了算法 5 中这一优化的伪代码。

成本分析。我们分析算法 4 中 *SeilInsert* 的成本。假设要插入 n 个向量。对 *assigns* 进行排序时，我们可以使用含有 *nlist* 个桶的桶排序，分两趟完成，耗时为 $O(n)$。在两个 for 循环中，*CountItemsWSameCell* 会遍历 *assigns* 数组中的每个项，产生 $O(n)$ 的成本。将 n 向量项及其 PQ 编码和向量 ID 打包到 *appendBlockInSharedCell* 和 *appendItemInMiscArea* 中的块，成本为 $O(n \cdot D)$。其余操作则针对每个单元执行。由于单元数量有上界 n，因此其成本有上界 $O(n)$。因此，*SeilInsert* 的成本为 $O(n \cdot D)$，主要由将向量项打包到块中的操作决定。我们的 SEIL 优化避免了为共享单元冗余存储块，从而降低了这项成本的常数因子。

接下来，我们分析算法1中 *AddVector* 的开销。该算法执行完整的插入流程，涉及 RAIR 和 SEIL。如第 4.2 节所述，每个向量的 *RairAssign* 开销为 $O((sublinear(nlist) + N_CANDS)D)$。对于 n 个向量，其开销为 $O((sublinear(nlist) + N_CANDS)nD)$。for 循环 *AddVector*中的其余操作（包括对向量进行 PQ 编码）需要 $O(n \cdot D)$时间。*SeilInsert*需要 $O(n \cdot D)$时间。因此，总体开销为 $O((sublinear(nlist) + N_CANDS)nD)$。

在 SIFT1M 数据集上，将所有数据向量插入 RAIRS 索引需要 15.8 秒，而单次分配的 IVFPQfs 基线需要 14.0 秒，HNSW 索引则需要 136.0 秒。RAIRS 和 IVFPQfs 的训练时间均为 13.3 秒。因此，RAIRS 和 IVFPQfs 的索引构建时间分别为 29.1 秒和 27.3 秒。尽管 RAIRS 的索引构建速度比 IVFPQfs 慢 6.6%，但仍远快于 HNSW。因此，我们认为 RAIRS 带来的额外构建开销在实际可接受范围内。关于插入的更多结果见第6.2节。

对于 *SeilSearch* 算法 5，设 $n_{vec_selected}$ 为选定列表中的向量总数。那么，*SeilSearch* 的最坏情况开销为 $O(n_{vec_selected}D)$。设 n_{vec_shared} 为两个选定列表所共享的单元块中的向量数。那么，SEIL优化的单元级去重将开销降低至$O((n_{vec_selected} - n_{vec_shared})D)$。

表2：实验中使用的数据集。

数据集	距离	#Dim	#项目数	#查询	Size
SIFT1M [30]	欧几里得距离	128	1,000,000	10,000	488MB
SIFT1B [30]	欧几里得距离	128	1,000,000,000	10,000	477GB
MSong [36]	欧几里得距离	420	994,185	1,000	1.56GB
GIST [30]	欧几里得距离	960	1,000,000	1,000	3.58GB
OpenAI [69]	欧几里得距离	1536	5,000,000	1,000	28.61GB
T2I [8]	内积	200	10,000,000	100,000	7.45GB

6 评估

我们首先在第6.1节介绍实验设置。接下来，我们在第6.2节报告 RAIRS 的整体性能，然后在第 6.3 节深入分析各项技术和算法参数。最后，我们在第 6.4 节研究 SEIL优化列表布局的灵活性，将其应用于 SOAR。

6.1 实验设置

机器配置。 我们在一台服务器上进行所有实验，该服务器配备 Intel(R) Xeon(R) Platinum 8360Y CPU（2.40GHz、36 个核心、每个核心 64KB L1 缓存、每个核心 1.25MB L2 缓存、共享 54MB L3 缓存）和 1TB 3200MT/s DDR4 内存，运行 Cen- tOS 7.9.2009。该 CPU 同时支持 AVX2 和 AVX-512 SIMD 指令[1]。C/C++ 代码使用 GCC 9.3.1 并通过 -O3 进行编译。

实现。 我们基于 Faiss v1.8.0 [18]实现 RAIRS索引，并采用 OpenBLAS v0.3.3 提供线性代数支持。RAIRS 由 Faiss 的 IndexIVF 类的多个子类实现。它通过 Faiss 的公共 API 与 Faiss 进行交互。我们编写了约 2,200 行代码。

在实验中，默认情况下，RAIR 分配和 SEIL 查询例程均以向量批次的方式执行。对于使用 *AddVectors* 进行的索引构建，会将所有数据向量作为一个批次提供，用于构建 RAIRS索引，该过程会执行 RAIR 分配并构建 SEIL 优化列表。对于查询处理，每个线程会并行处理一批查询向量，以最大限度地提高吞吐量。我们会批量执行*FindNearestLists*函数，为查询批次中的所有查询向量检索最近列表。然后，按照第 5.3 节所述，对查询批次进行缓存优化。我们会扫描每个列表，处理与之关联的所有查询，这样当前列表便可保留在 CPU 缓存中，从而提升整体性能。

为支持删除，FAISS IVF 索引维护一张映射表，将每个向量 ID 映射到该向量的列表 ID 和列表内位置。在 RAIRS 中，我们修改映射表，使每个向量 ID 最多包含 2 个列表 ID 和列表内位置。给定要删除的向量 ID，如果其列表内位置位于共享块中，我们就将打包块中的相应 ID 设为无效 ID。如果该位置位于杂项块中，我们就用杂项块中的最后一个向量 ID 替换此条目。由于每个向量最多会分配到两个列表，RAIRS 在删除时修改的条目最多是基线 IVFPQfs 的两倍。

对比方案。 我们比较以下流行的近似最近邻搜索索引构建方法和冗余分配策略：

[1]FAISS v1.8.0 中的 IndexIVFPQFastScan 类使用 AVX2 256 位 SIMD 指令，但 FAISS 的编译会利用 AVX-512 指令生成 libfaiss_avx512.so。我们的实验使用 libfaiss_avx512.so。在我们的实验中，不会发生降频。

- **倒排文件** [47]：Faiss 1.8.0 中的 IVFFlat 类。这是普通的 IVF 索引，其列表遍历会执行精确的距离计算。对于每个数据集，我们使用与 RAIRS 相同的 IVF 参数。
- **HNSW** [32]：Faiss 中带有 HNSW 索引的 IVF 类，这是一个实现了近似最近邻搜索索引。除 M 外，我们遵循其原始工作中的默认设置 [39]（例如，$efConstruction = 500$）。我们设置为 $M = 32$，这是 Faiss 的默认设置，因为与原始工作中的 $M = 16$ 相比，它能实现更好的性能。
- **IVFPQfs**：实现了倒排文件 PQ 快速扫描。为提高召回率，我们添加了一个精炼层。IVFPQfs 与 RAIRS 之间的主要区别在于 RAIR 分配和 SEIL 布局。IVFPQfs 执行基线单次分配。我们使用与 RAIRS 相同的参数。
- **NaïveRA** [13]：我们实现了朴素的冗余分配策略。它采用带精炼结构的 IVF-PQ 快速扫描，并使用与 RAIRS 相同的参数。默认情况下不启用 SEIL。
- **SOARL2** [50]：我们将 NaïveRA 的冗余分配策略替换为 SOAR。请注意，SOAR 最初是为内积距离设计的。在这里，我们直接将 SOAR 应用于欧几里得空间中的 L2 距离。
- **RAIR and SRAIR**：不采用 SEIL 列表布局优化的 RAIR 和严格 RAIR（参见第 4.2 节）。
- **RAIRS and SRAIRS**：采用 SEIL 列表布局优化的 RAIR 和严格 RAIR。

线程级并行由 Faiss 库处理。我们确保所有方案均使用相同数量的线程和相同的并行化方案。不使用超线程。对于采用 IVF-PQ 快速扫描 + 精炼结构的每个索引，我们始终对查询批次采用缓存优化。

数据集。 在实验中，我们使用了以下具有代表性的真实世界数据集，涵盖了多种向量维度、向量数量和应用场景。数据集的特征总结于表2中。

- **SIFT** [30]：SIFT（尺度不变特征变换）描述符源自一个图像数据集。我们使用两个 SIFT 数据集，即 SIFT1M 和 SIFT1B，分别包含 100 万和 10 亿个向量。在 SIFT1B 中，原始描述符以 8 位整数存储；为与其他数据集保持一致，我们在预处理中将其转换为 32位浮点数。
- **MSong** [36]：百万歌曲数据集包含 100 万首流行音乐曲目的特征。
- **GIST** [30]：GIST 描述符由一个图像数据集生成，用于捕捉场景的空间结构。
- **OpenAI**：此 OpenAI 嵌入数据集由 Common Crawl 中开源的 C4 数据集生成。我们使用 VectorDBBench 的命令行工具下载该数据集 [69]，度量类型为 L2 距离。
- **T2I** [8]：Yandex 文本到图像数据集包含作为数据向量的图像嵌入，以及作为查询的文本嵌入。

由于 RAIRS 面向欧几里得空间，我们的主要实验使用 SIFT、MSong、GIST 和 OpenAI。与其他数据集不同，T2I 的距离度量是内积。我们使用 T2I 研究 SEIL 对原始 SOAR 的适用性，详见第 6.4 节。

表 3：在 SOARL2 下第二选择质心与在 AIR 下相同的向量百分比。

	SIFT1M	SIFT1B	MSong	GIST	OpenAI
SOARL2	95.14%	72.10%	93.93%	91.55%	93.40%

参数设置。默认情况下，我们设置 $nlist = 1024$，但数据集 OpenAI [69] ($nlist = 2048$)、T2I [8] ($nlist = 3172$) 和 SIFT1B [30]($nlist = 32768$) 除外。这些 $nlist$ 值接近每个数据集的 $O(\sqrt{\#Item})$，这与 Faiss库 [17]中的建议一致。对于 PQ 编码，我们将子组数量设置为 $M_{PQ} = \frac{Dim}{2}$，并将每个子组的码字位长设置为 $nbits_{PQ} = 4$。在精炼模块中，我们为 top-1 和 top-10 查询设置 $K_FACTOR = 10$。（10 是 ANN-Benchmark 结果 [5]中 faiss-ivfpqfs 实验常用的设置。）对于 top-100 查询，我们将 $K_FACTOR = 4$ 设置为，以平衡列表遍历和向量精化所需的时间。对于 RAIRS，我们根据第 6.3 节对参数设置的实验研究，设置 $\lambda = 0.5$ 和 $N_CANDS = 10$。

在查询时，我们会改变搜索参数（例如，倒排文件索引中的 $nprobe$ 和 HNSW 中的 $efSearch$），以实现查询速度与搜索质量之间不同的权衡；这些权衡对应于实验结果中所报告曲线上的各个点。

性能指标。1)**召回率-QPS**。由于近似最近邻搜索本质上需要在准确率与速度之间权衡，因此仅凭吞吐量无法全面反映其性能。因此，我们报告召回率-QPS 曲线，以体现检索质量与处理效率之间的平衡。对于 top-K 查询，召回率 $k@K$ 表示查询结果中真实 top-K 最近邻所占的平均比例。查询吞吐量以 QPS（每秒查询数）表示。2) **召回率-DCO**。近似最近邻搜索的查询处理中一项重要开销是距离计算 [21]。DCO 表示每次查询的距离计算操作数。我们使用 DCO 来了解冗余分配策略的有效性。与召回率-QPS 曲线类似，构建召回率-DCO曲线时，会改变搜索参数（例如，$nprobe$），并绘制召回率 $k@K$ 与相应 DCO 之间的关系。

6.2 整体性能

与常用近似最近邻搜索方法的比较。图 7a 展示了 IVF、HNSW、IVFPQfs 和 RAIRS 在五个具有代表性的真实世界数据集上的召回率-QPS 曲线。每行子图对应一个数据集。左列报告 top-1 查询的结果，右列展示 top-10 查询的性能。在每个图中，越靠近右上角，性能越好。对于每个搜索参数及其对应的召回率，我们运行实验 10 次并报告平均每秒查询数。

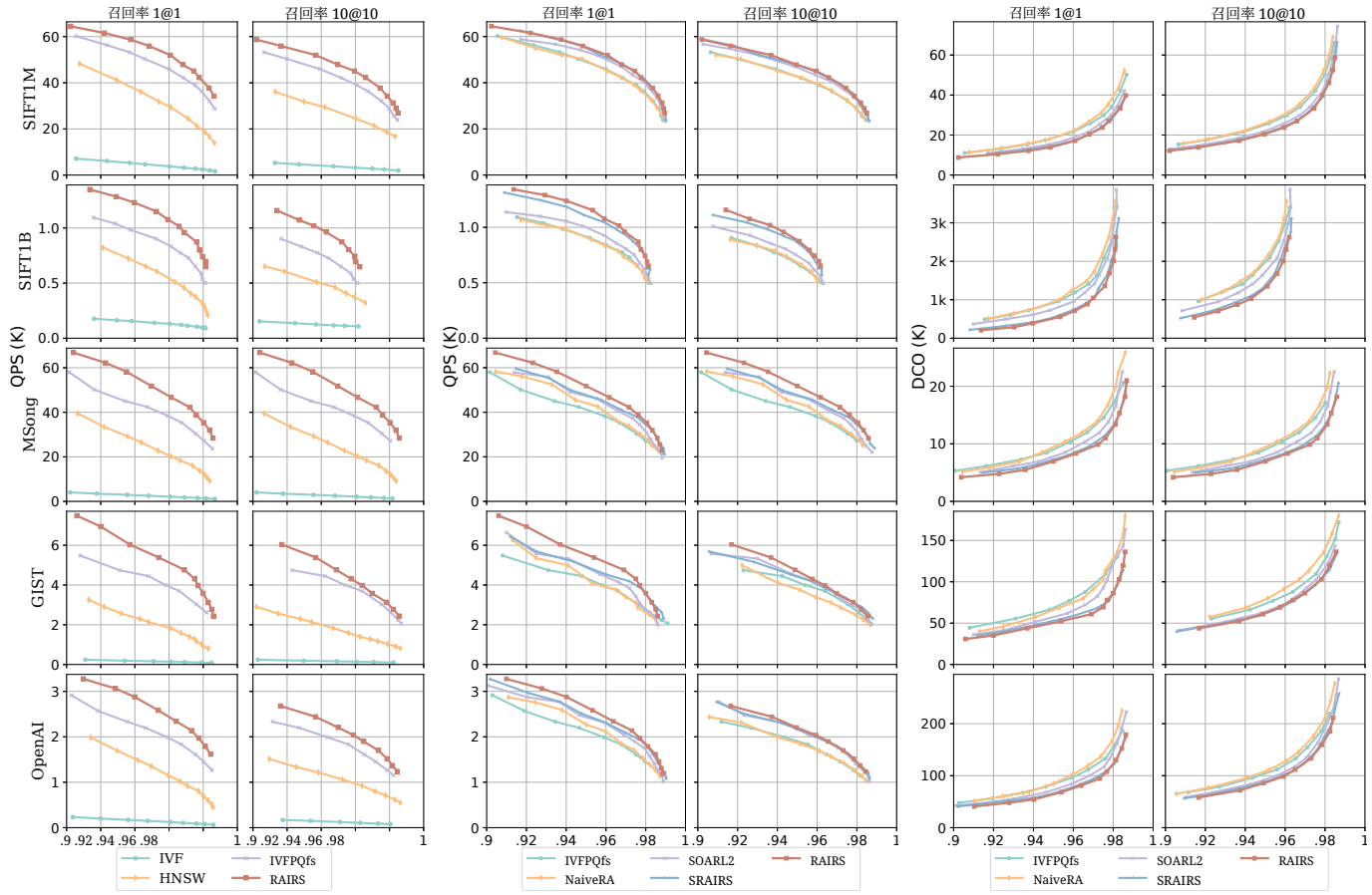
如图 7a 所示，IVFPQfs 和 RAIRS 的性能显著高于 IVF 和 HNSW。这一性能优势主要来自 PQ 快速扫描 [3] 的 SIMD 加速，该加速在列表遍历过程中处理打包块。这种方法比对每个单个向量进行距离计算快得多。此外，精炼层弥补了 PQ 编码导致的准确率下降。

在各种近似最近邻搜索方法中，我们提出的 RAIRS 性能最佳。它结合了 RAIR 冗余分配和 SEIL 优化列表布局，以提升基于倒排文件的近似最近邻搜索性能。与性能排名第二的方法（即 IVFPQfs）相比，RAIRS 在所有真实世界数据集上均能以相近的召回率，将查询吞吐量提升至最高 1.33 倍。

各种分配策略的比较。图 7b 展示了五种分配策略在五个真实世界数据集上针对 top-1 和 top-10 查询的召回率-QPS 曲线。这五种方案均基于带精化的 IVF-PQ 快速扫描。根据图 7b，我们得出以下观察结果。(1) 与基线单次分配（即 IVFPQfs）相比，采用 NaïveRA 的冗余分配并没有更好，尤其是在高召回率下。对于 GIST 上的 top-10 查询，NaïveRA 的表现实际上不如 IVFPQfs。这表明优化列表选择策略的重要性。(2) 在所有冗余分配策略中，RAIRS 在所有数据集上的表现最佳。在召回率为 0.95 时，与 IVFPQfs、NaïveRA 和 SOARL2 相比，RAIRS 的吞吐量分别提升了 1.07–1.33 倍、1.11–1.32 倍和 1.01–1.23 倍。(3) 对每个向量严格进行两次分配的 SRAIRS，在 SIFT1M 和 SIFT1B 上与 RAIRS 表现相当，但在 MSong、GIST 和 OpenAI 上不如 RAIRS。这意味着，当首次选择的列表与第二次选择的列表相同时，最好只存储一次该向量项。(4) 在大多数情况下，RAIRS 显著优于 SOARL2，因为与 SOAR 不同，RAIRS 针对欧几里得空间进行了优化。在 SIFT1M 上的召回率 1@1，以及 SIFT1M 和 OpenAI 上的召回率 10@10 场景中，SOARL2 的表现与 RAIRS 相当。AIR 倾向于选择一个 c'，使得 $r' = c' - x$ 更接近 $r = c - x$ 的逆向量（即 θ 越接近 180 度越好）。然而，具体取决于给定的数据集和所考虑的向量，实际 c' 的角度 θ 可能远小于 180 度，从而产生与 SOARL2 相似的分配。表 3 报告了在 SOARL2 下，第二选择质心与 AIR 下第二选择相匹配的向量所占比例。我们发现，AIR 和 SOARL2 对 72.10%–95.14% 的向量采用了相同的分配。在上述 SOARL2 与 RAIRS 表现相近的情况（即 SIFT1M 和 OpenAI）中，第二选择匹配的比例较高。从另一个角度看，4.86%–27.90% 的分配差异带来了 RAIRS 相较于 SOARL2 的性能优势。一般而言，差异越大，RAIRS 的潜在优势也越大。

通过 DCO 理解分配策略。图 7c 展示了各种分配策略的召回率-DCO 曲线。曲线越靠近右下角，效果越好。召回率-DCO 曲线与图 7b 中的召回率-QPS 曲线呈现出相似的趋势。由于这些方案都基于带精化的 IVF-PQ 快速扫描，因此查询吞吐量的差异主要源于计算距离数量的差异。在召回率为 0.95 时，与 IVFPQfs、NaïveRA 和 SOARL2 相比，RAIRS 将 DCO 分别降低了 0.64–0.83 倍、0.62–0.78 倍和 0.73–0.99 倍。

请注意，RAIRS 的优势不仅来自更高的召回率，也来自更低的 DCO，这一点已由图 7c 证实。RAIRS 可以在 DCO 相同的情况下实现更高的召回率，也可以在 DCO 更低的情况下实现相同的召回率（这是由更低的 $nprobe$ 导致的）。图 8 展示了改变 $nprobe$ 时的召回率。比较实现相近召回率所需的 $nprobe$，我们可以看到，SRAIRS 和 RAIRS 的 $nprobe$ 分别为基线单次分配 IVFPQfs 设置的 42.3%–46.5% 和 48.1%–53.1%。对于 RAIRS，由于部分向量仅被分配一次，要达到与 SRAIRS 相当的召回率，可能需要稍大的 $nprobe$。



（a）与常用近似最近邻搜索方法的比较 （b）不同分配策略 (c) 分配策略的 DCO

图 7：整体近似最近邻搜索性能。

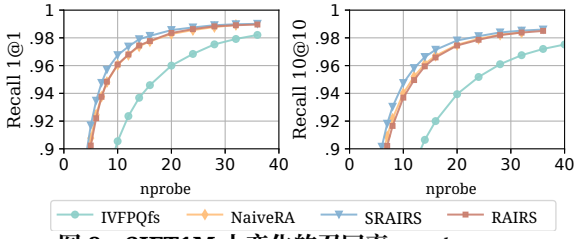


图 8：SIFT1M 上变化的召回率 *nprobe*。

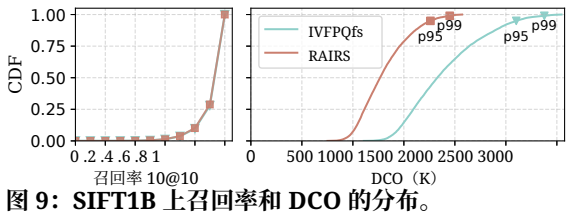


图 9：SIFT1B 上召回率和 DCO 的分布。

给定搜索参数，对于给定的近似最近邻搜索索引上的一组查询，DCO 是确定性的。相比之下，QPS 的稳定性较低，会受到运行时变化的影响。因此，对于基于带精化的 IVF-PQ 快速扫描的方案，在评估的其余部分，我们以 DCO 展示近似最近邻搜索性能。

召回率和 DCO 的分布。对于图 7 中 SIFT1B + 0.95 召回率实验，我们计算 top-10 搜索实验中 10,000 个查询各自的召回率和 DCO。我们在图 9 中绘制了召回率和 DCO 的累积分布函数（CDF）。从图中可以看出，召回率 CDF 曲线几乎重合，但 RAIRS 的 DCO 曲线明显位于 IVFPQfs 的 DCO 曲线左侧。这意味着，与 IVFPQfs 相比，RAIRS 在达到相近召回率的同时，显著减少了几乎所有查询的 DCO。此外，召回率方差较低；超过 89% 的查询达到 0.8–1.0 的召回率。RAIRS 的 p99 DCO 是平均值的 1.50 倍。不同查询之间的 DCO 方差适中。

Top-100 查询的性能。图 10 比较了 IVFPQfs、NaïveRA、SOARL2 和 RAIRS 在 top-100 查询上的召回率-DCO 曲线。可以看出，RAIRS 在所有数据集上都取得了最佳性能，这与图 7c 中 top-1 和 top-10 查询的结果一致。

一次处理一个查询时的性能。在这项实验中，我们一次运行一个查询，以了解未对查询批次进行缓存优化时的单线程查询延迟。图 11 展示了 SIFT1M 和 GIST1M 的查询延迟-召回率曲线。可以看到，在所有单次分配和二重分配策略中，RAIRS 的延迟最低

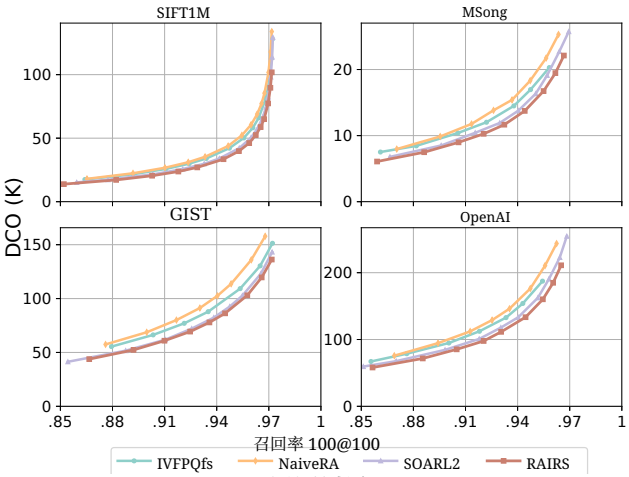


图 10：Top-100 查询的性能。

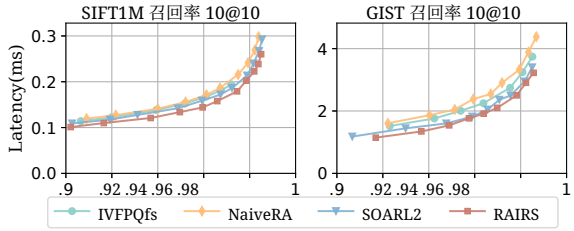


图 11：每次执行一个查询时的性能。

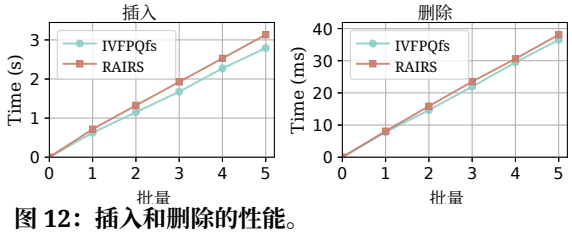


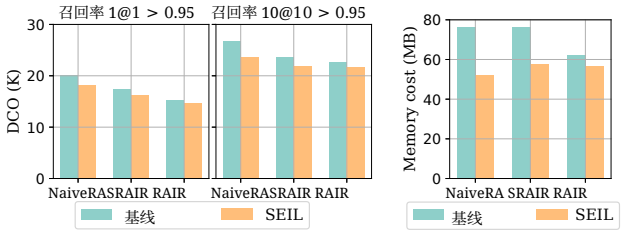
图 12：插入和删除的性能。

。RAIRS 带来的收益与批量查询场景中的情况相似。

插入和删除的性能。图 12 展示了 RAIRS 和 IVFPQfs 在 SIFT1M 上进行向量插入和删除的性能。对于插入，我们先用 800,000 个向量构建索引，然后执行五个插入批次，每批插入 40,000 个新向量。对于删除，我们用完整数据集构建索引，然后执行五批删除，每批删除 40,000 个向量。与基线 IVFPQfs 相比，RAIRS 的插入吞吐量和删除吞吐量分别低 12.2% 和 4.4%。这是因为每个向量最多被分配到两个列表中，RAIRS 修改的条目最多是 IVFPQfs 的两倍。请注意，近似最近邻搜索通常比向量插入和删除操作频繁得多。尽管插入和删除会产生少量额外开销，RAIRS 仍能实现更高的近似最近邻搜索查询性能。

6.3 深入分析

RAIR 和 SEIL 的消融研究。图 13a 比较了在 SIFT1M 数据集上，NaïveRA、SRAIR 和 RAIR 在采用与未采用 SEIL 列表布局优化时的 DCO。对于每种对比方案，我们报告查询召回率刚刚超过 95% 时的 DCO。在图中，DCO 越低越好。



（a）近似最近邻搜索性能 **（b）内存开销**

图 13：RAIR 和 SEIL 在 SIFT1M 上的消融研究。

表 4：欧几里得距离数据集的内存开销。

数据集	IVFPQfs	NaïveRA	NaïveRA+SEIL	RAIR	RAIRS
SIFT1M	38.2 MB	76.4 MB	52.0 MB	62.5MB	56.25 MB
SIFT1B	37.3 GB	74.6 GB	42.9 GB	69.7 GB	43.3 GB
MSong	107.2 MB	214.5 MB	145.0 MB	162.8MB	152.4 MB
GIST	240.6 MB	478.5 MB	325.0 MB	403.3MB	358.7 MB
OpenAI	1.83GB	3.66 GB	2.39 GB	3.15 GB	2.53 GB

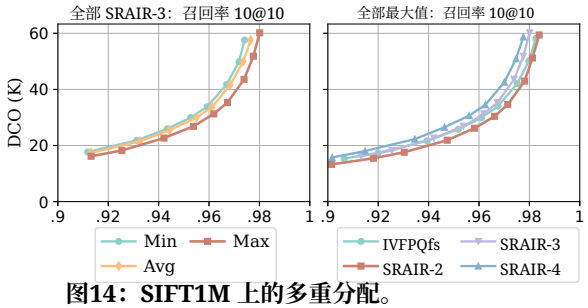


图14：SIFT1M 上的多重分配。

可以看到，无论是否采用 SEIL，RAIR 的表现都优于 NaïveRA。未采用 SEIL 时，与 NaïveRA 相比，RAIR 对 top-1 和 top-10 查询的 DCO 分别降低了 24.0% 和 14.9%。采用 SEIL 时，DCO 分别降低了 18.8% 和 8.0%。

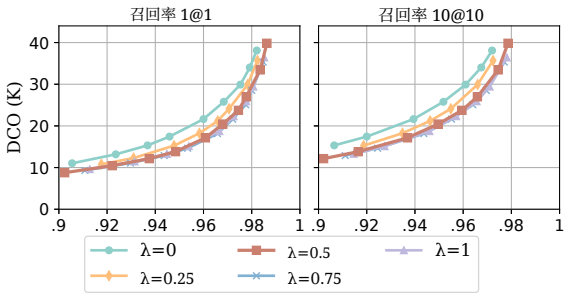
此外，SEIL 能有效减少冗余的距离计算。对于 top-1 和 top-10 查询，SEIL 可使 NaïveRA、SRAIR 和 RAIR 的 DCO 降低 4.1%–12.0%。

最后，对比 RAIR 和 SRAIR 可以发现，RAIR 的 DCO 更低。当第一个和第二个分配列表相同时，RAIR 会执行单次分配，而 SRAIR 则严格地从其余列表中选择第二个列表来分配向量。图 13a 中的结果表明，这种严格策略并非最优。

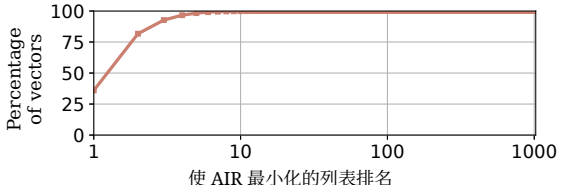
内存开销。图 13b 展示了 SIFT1M 上 NaïveRA、SRAIR 和 RAIR 在启用与未启用 SEIL 时的内存开销。表 4 报告了基线 IVPQfs 和针对五个代表性数据集的四种冗余分配方案的内存开销。由于精炼层相同，我们省略精炼层，仅报告 IVF-PQ 模块占用的空间。

我们可以看到，由于冗余分配，NaïveRA 的内存占用是基线 IVPQfs 的两倍。SEIL 仅存储一次大型单元的共享块，从而显著降低冗余分配的内存开销。总体而言，对于 NaïveRA、SRAIR 和 RAIR，SEIL 可将内存开销降低 6.4%–42.5%。

请注意，精炼层存储了所有向量数据。与精炼层相比，基线 IVF-PQ 模块占用的空间为 6.4%–7.5%，而 RAIRS 额外占用的空间为 1.2%–3.7%。因此，RAIR 以略微增加空间为代价，显著提升了近似最近邻搜索性能。



(a) 改变 λ时的召回率-DCO曲线



(b) 使 AIR 最小化的列表排名分布

图15：参数研究（SIFT1M）。

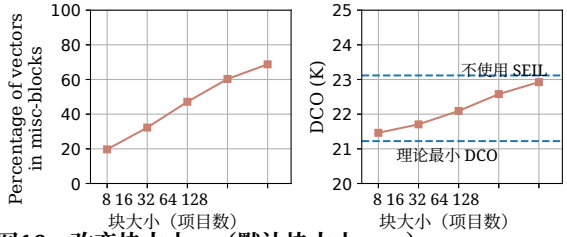


图16：改变块大小。（默认块大小 = 32）

多重分配。 图14左侧比较了 SIFT1M 上用于三重分配的三种聚合函数（即 *max*、*min*和 *avg*）。*max*的 DCO 最低。图14右侧比较了单次分配（即 IVFPQfs）、二重分配、三重分配和四重分配。我们选择 *max*聚合函数。由于 SEIL 是为二重分配设计的，因此将其禁用。我们采用严格策略（即 SRAIR），因为 RAIR 本质上模糊了多重分配之间的区别。从图中可以看出，SRAIR-2 的性能最佳。这一结果表明，两次以上的分配并无必要。

针对 λ的参数研究。图15a展示了 RAIRS 在 SIFT1M 上的召回率-DCO 曲线，其中 λ 从 0 变化到 1。随着 λ 增大，曲线向右下方移动，表明性能有所提升。当 λ 达到 0.5 时，曲线不再改善。因此，我们将 λ 的默认值设为 0.5。

参数研究 N_CANDS。我们修改了算法 3，使其*RairAssign* 按照列表质心与待插入向量之间距离的升序，检索所有列表作为候选项，并计算使 AIR 度量最小的列表。这得出了 AIR 应分配的排序列表中的真实排名。图15b展示了 SIFT1M 中所有向量真实排名的累积分布函数（CDF）。我们发现，在 99.95% 的情况下，真实排名为 ≤ 10。这意味着，考虑最近的前 10 个列表即可在大多数情况下实现正确分配。因此，我们默认设置为 $N_CANDS = 10$。

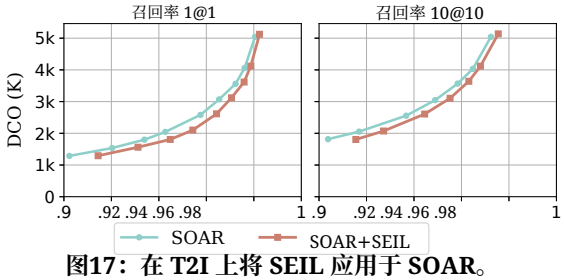


图17：在 T2I 上将 SEIL 应用于 SOAR。

PQ 块大小的参数研究。 为了解块大小对 SEIL 的影响，我们改变块大小，研究杂项块中向量数量的变化，并计算给定 *nprobe*时产生的 DCO 数量。如图16所示，随着块大小增大，杂项块中的向量会增多。这是因为大型单元（其大小 ≥ 块大小）的数量减少了。SEIL 会为杂项块中的向量执行更多冗余 DCO。

6.4 将 SEIL 应用于 SOAR

我们将 SEIL 应用于针对内积距离的 SOAR。图17比较了 SOAR 和 SOAR+SEIL 在 T2I 上的召回率-DCO曲线，T2I 使用内积距离作为距离度量。

如图17所示，SEIL 显著降低了 SOAR 的 DCO。这一结果表明，SEIL 列表布局优化适用于各种冗余分配策略和距离度量。

7 相关工作

近似最近邻搜索方法。 向量数据库和通用数据库系统均已实现对近似最近邻搜索的支持 59, 63, 67] [23, 34, 52, 54,]。

现有的近似最近邻搜索方法分为四类：基于树 [10, 33, 41, 42, 64]、、基于哈希 [2, 9, 35, 38, 53, 56, 60]、基于图 [20, 28, 39, 44, 45, 49, 51, 68], [7, 26, 和基于聚类的 27, 47] [47], 方法。基于倒排文件的近似最近邻搜索是基于聚类方法中的代表类型，也是应用最广泛、性能最高的近似最近邻搜索方法之一。本文主要关注基于倒排文件的近似最近邻搜索。

倒排文件近似最近邻搜索方法的优化途径。 优化基于倒排文件的近似最近邻搜索主要有四种途径：1）量化，2）距离计算优化，3）硬件加速，以及 4）冗余分配。下文将回顾第 1 至第 3 种途径的相关工作。冗余分配是本文的重点。我们已讨论并通过实验比较了现有的冗余分配策略。

途径 *1*：量化。量化技术被广泛用于降低存储开销并加速相似性搜索。现有量化方法可按维度分组的粒度分为：单维量化（包括 VA_File [58], ITQ [24], LVQ [1], 和 RabitQ [22]）、全局量化（包括 AQ [6] 和 LSQ [40]），以及乘积量化（包括 PQ [31], OPQ [23], PolysemousCodes [19], NEQ [15], DeltaPQ [55], VAQ [43], QAQ [66], 和 SparCode [48]）。

方法二：基于采样的距离计算。鉴于维度较高，距离计算往往占据近似最近邻搜索查询响应时间的主要部分 [21]。ADSampling [21] 利用 Johnson–Lindenstrauss（JL）引理，并支持提前终止

在距离比较过程中对部分维度进行采样，从而有效减少参与计算的维度数量。

方法 *3*：硬件加速。人们已研究了多种硬件特性在基于倒排文件的近似最近邻搜索中的应用。SPANN [13,62] 利用固态硬盘来减少内存占用并提供数据持久性。GPU [32] 和现场可编程门阵列 [16, 65] 已被证明能显著加速近似最近邻搜索处理。PQ 快速扫描 [3],Bolt [11], 和 Quicker ADC [4] 利用 SIMD 指令加速距离计算。

与我们工作的关系：总体而言，我们提出的想法（即 RAIR 和 SEIL）与上述三种优化方法互为补充。本文具体采用结合 PQ 量化、精化和 PQ 快速扫描的倒排文件作为基线，这是实践中性能最佳的基于倒排文件的方法之一。我们针对 PQ 快速扫描对 SEIL 优化进行了调整。

8 结论

总之，本文提出了 RAIRS，将两种优化技术（即 RAIR 和 SEIL）结合起来，用于基于倒排文件的近似最近邻搜索。RAIR 利用冗余分配，提升那些无法通过首批分配列表获得良好服务的查询的性能。我们正式推导了 AIR列表选择指标。此外，SEIL 优化列表布局，并利用共享单元减少冗余距离计算。在具有代表性的真实世界数据集上取得的实验结果证实，RAIR 和 SEIL 能够有效降低 DCO，并提升基于 IVF 的近似最近邻搜索性能。

参考文献

[1] Cecilia Aguerrebere、Ishwar Singh Bhati、Mark Hildebrand、Mariano Tepper 和 Theodore Willke。2023。眨眼之间的相似性搜索：压缩索引。*VLDB*基金会会刊 16, 11 (2023), 3433–3446。[2] Alexandr Andoni 和 Piotr Indyk。2008。高维近似最近邻的近乎最优哈希算法。*ACM*通讯 51, 1 (2008), 117–122。[3] Fabien André、Anne-Marie Kermarrec 和 Nicolas Le Scouarnec。2015。缓存局部性还不够：使用乘积量化快速扫描实现高性能最近邻搜索。*VLDB*基金会会刊 9, 4 (2015), 288–299。[4] Fabien André、Anne-Marie Kermarrec 和 Nicolas Le Scouarnec。2019。Quicker ADC：利用 SIMD 释放乘积量化的隐藏潜力。*IEEE*模式分析与机器智能汇刊 43, 5 (2019), 1666–1677。[5] Martin Aumüller、Erik Bernhardsson 和 Alexander Faithfull。2020。ANN-Benchmarks：近似最近邻算法的基准测试工具。信息系统 87 (2020), 101374。[6] Artem Babenko 和 Victor Lempitsky。2014。用于极限向量压缩的加性量化。载于*IEEE*计算机视觉与模式识别会议论文集。931–938。[7] Artem Babenko 和 Victor Lempitsky。2014。倒排多索引。*IEEE*模式分析与机器智能汇刊 37, 6 (2014), 1247–1260。[8] Dmitry Baranchuk 和 Artem Babenko。2021。用于十亿规模相似性搜索的文本到图像数据集。h t t p s:// r es earch.yandex.com/datasets/text-to-image-dataset-for-billion-scale-similarity-search [9] Mayank Bawa、Tyson Condie 和 Prasanna Ganesan。2005。LSH森林：用于相似性搜索的自调优索引。载于第*14*届万维网国际会议论文集。651–660。[10] Jon Louis Bentley。1975。用于关联搜索的多维二叉搜索树。*ACM*通讯 18, 9 (1975), 509–517。[11] Davis W Blalock 和 John V Guttag。2017。Bolt：使用快速向量压缩加速数据挖掘。载于第*23*届*ACM SIGKDD* 知识发现与数据挖掘国际会议论文集。727–735。[12] Sebastian Borgeaud、Arthur Mensch、Jordan Hoffmann、Trevor Cai、Eliza Rutherford、Katie Millican、George Bm Van Den Driessche、Jean-Baptiste Lespiau、Bogdan Damoc、Aidan Clark 等。2022。通过从数万亿个词元中检索内容来改进语言模型。载于国际机器学习会议。PMLR，2206–2240。[13] Qi Chen、Bing Zhao、Haidong Wang、Mingqin Li、Chuanjie Liu、Zengzhong Li、Mao Yang 和 Jingdong Wang。2021。SPANN：高效的十亿规模近似最近邻域搜索。《神经信息处理系统进展》34（2021），5199–5212。[14] Thomas Cover 和 Peter Hart。1967。最近邻模式分类。《IEEE信息论汇刊》13，1（1967），21–27。[15] Xinyan Dai、Xiao Yan、Kelvin KW Ng、Jiu Liu 和 James Cheng。2020。范数显式量化：改进用于最大内积搜索的向量量化。载于《AAAI人工智能会议论文集》，第34卷。51–58。[16] Dimitrios Danopoulos、Christoforos Kachris 和 Dimitrios Soudris。2019。高维向量近似kNN索引的FPGA加速。载于《2019年第14届片上可重构通信中心系统国际研讨会（ReCoSoC）》。IEEE，59–65。[17] Matthijs Douze。2024。索引选择指南。https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index[18] Matthijs Douze、Alexandr Guzhva、Chengqi Deng、Jeff Johnson、Gergely Szilvasy、Pierre-Emmanuel Mazaré、Maria Lomeli、Lucas Hosseini 和 Herve Jégou。2024。Faiss库。（2024 2401 08281 [ ]）。arXiv:[19]. cs.LG Matthijs Douze、Hervé Jégou 和 Florent Perronnin。2016。多义码。载于《计算机视觉—ECCV 2016：第14届欧洲会议，阿姆斯特丹，荷兰，2016年10月11日至14日，论文集，第二部分14》。施普林格，785–801。[20] Cong Fu、Chao Xiang、Changxu Wang 和 Deng Cai。2019。基于导航扩散图的快速近似最近邻搜索。《VLDB基金会学报》12，5（2019），461–474。[21] Jianyang Gao 和 Cheng Long。2023。高维近似最近邻搜索：可靠且高效的距离比较操作。《ACM数据管理学报》1，2（2023），1–27。[22] Jianyang Gao 和 Cheng Long。2024。RaBitQ：以理论误差界量化高维向量，用于近似最近邻搜索。《ACM数据管理学报》2，3（2024），1–27。[23] Tiezheng Ge、Kaiming He、Qifa Ke 和 Jian Sun。2013。用于近似最近邻搜索的优化乘积量化。载于《2013年IEEE计算机视觉与模式识别会议论文集》。2946–2953。[24] Yunchao Gong、Svetlana Lazebnik、Albert Gordo 和 Florent Perronnin。2012。迭代量化：一种用于大规模图像检索二进制码学习的普罗克拉斯特斯方法。《IEEE模式分析与机器智能汇刊》35，12（2012），2916–2929。[25] Rentong Guo、Xiaofan Luan、Long Xiang、Xiao Yan、Xiaomeng Yi、Jigao Luo、Qianya Cheng、Weizhi Xu、Jiarui Luo、Frank Liu 等。2022。Manu：云原生向量数据库管理系统。《VLDB基金会学报》15，12（2022），3548–3561。[26] Ruiqi Guo、Philip Sun、Erik Lindgren、Quan Geng、David Simcha、Felix Chern 和 Sanjiv Kumar。2020。利用各向异性向量量化加速大规模推理。载于《国际机器学习会议》。PMLR，3887–3896。[27] Gaurav Gupta、Tharun Medini、Anshumali Shrivastava 和 Alexander J Smola。2022。Bliss：采用迭代式重新分区的十亿规模索引。载于《第28届ACM SIGKDD知识发现与数据挖掘会议论文集》。486–495。[28] Ben Harwood 和 Tom Drummond。2016。FANNG：快速近似最近邻图。载于《IEEE计算机视觉与模式识别会议论文集》。5713–5722。[29] Piotr Indyk 和 Rajeev Motwani。1998。近似近邻：迈向消除维度灾难。载于《第三十届ACM计算理论年会论文集》。604–613。[30] Herve Jegou 和 Laurent Amsaleg。2010。近似最近邻搜索数据集。http://corpus-texmex.irisa.fr [31] Herve Jegou、Matthijs Douze 和 Cordelia Schmid。2010。用于最近邻搜索的乘积量化。《IEEE模式分析与机器智能汇刊》33，1（2010），117–128。[32] Jeff Johnson、Matthijs Douze 和 Hervé Jégou。2019。利用GPU进行十亿规模相似性搜索。《IEEE大数据汇刊》7，3（2019），535–547。[33] Bastian Leibe、Krystian Mikolajczyk 和 Bernt Schiele。2006。目标类别识别中的高效聚类与匹配。。载于《BMVC》。789–798。[34] Jie Li、Haifeng Liu、Chuanghua Gui、Jianyu Chen、Zhenyuan Ni、Ning Wang 和 Yuan Chen。2018。京东电子商务平台实时视觉搜索系统的设计与实现。载于《第19届国际中间件会议工业分会论文集》。9–16。[35] Jinfeng Li、Xiao Yan、Jian Zhang、An Xu、James Cheng、Jie Liu、Kelvin KW Ng 和 Ti-chung Cheng。2018。一种通用且高效的学习哈希查询方法。载于《2018年国际数据管理会议论文集》。1333–1347。[36] Thomas Lidy。2015。百万歌曲数据集基准。https://www.ifs.tuwien.ac.at/mir/msd[37] Ying Liu、Dengsheng Zhang、Guojun Lu 和 Wei-Ying Ma。2007。基于高层语义的图像检索综述。《模式识别》40，1（2007），262–282。[38] Qin Lv、William Josephson、Zhe Wang、Moses Charikar 和 Kai Li。2007。多探针LSH：高维相似性搜索的高效索引构建。载于《第33届国际超大型数据库会议论文集》。950–961。

[39] Yu A Malkov 和 Dmitry A Yashunin。2020。使用分层可导航小世界图进行高效且稳健的近似最近邻搜索。《IEEE模式分析与机器智能汇刊》42，4（2020），824–836。[40] Julieta Martinez、Joris Clement、Holger H Hoos 和 James J Little。2016。重新审视加性量化。载于《计算机视觉—ECCV 2016：第14届欧洲会议，阿姆斯特丹，荷兰，2016年10月11日至14日，论文集，第二部分14》。施普林格，137–153。[41] Marius Muja 和 David G Lowe。2009。结合自动算法配置的快速近似近邻。。载于《国际计算机视觉理论与应用会议》，第2卷。2。[42] Marius Muja 和 David G Lowe。2014。面向高维数据的可扩展最近邻算法。《IEEE模式分析与机器智能汇刊》36，11（2014），2227–2240。[43] John Paparrizos、Ikraduya Edian、Chunwei Liu、Aaron J Elmore 和 Michael J Franklin。2022。通过方差感知量化实现快速自适应相似性搜索。载于《2022年IEEE第38届国际数据工程会议（ICDE）》。IEEE，2969–2983。[44] Yun Peng、Byron Choi、Tsz Nam Chan、Jianye Yang 和 Jianliang Xu。2023。多维数据库中的高效近似最近邻搜索。《ACM数据管理学报》1，1（2023），1–27。[45] Jie Ren、Minjia Zhang 和 Dong Li。2020。HM-ANN：异构内存上的高效十亿点近邻搜索。载于《第34届神经信息处理系统国际会议论文集》，第33卷。10672–10684。[46] J Ben Schafer、Dan Frankowski、Jon Herlocker 和 Shilad Sen。2007。协同过滤推荐系统。载于《自适应网络：网络个性化的方法与策略》。施普林格，291–324。[47] Sivic 和 Zisserman。2003。Video Google：一种用于视频中对象匹配的文本检索方法。载于《第九届IEEE国际计算机视觉会议论文集》。IEEE，1470–1477。[48] Liangcai Su、Fan Yan、Jieming Zhu、Xi Xiao、Haoyi Duan、Zhou Zhao、Zhenhua Dong 和 Ruiming Tang。2023。超越双塔匹配：学习稀疏可检索交叉交互以实现推荐。载于《第46届国际ACM SIGIR信息检索研究与发展会议论文集》。548–557。[49] Suhas Jayaram Subramanya、Devvrit、Rohan Kadekodi、Ravishankar Krishnaswamy 和 Harsha Vardhan Simhadri。2019。DiskANN：单节点上的快速、准确十亿点近邻搜索。载于《第33届神经信息处理系统国际会议论文集》。13766–13776。[50] Philip Sun、David Simcha、Dave Dopson、Ruiqi Guo 和 Sanjiv Kumar。2023。SOAR：改进近似最近邻搜索的索引构建。载于《第37届神经信息处理系统国际会议论文集》。3189–3204。[51] Godfried T Toussaint。1980。有限平面点集的相对邻域图。《模式识别》12，4（1980），261–268。[52] Jianguo Wang、Xiaomeng Yi、Rentong Guo、Hai Jin、Peng Xu、Shengjun Li、Xiangyu Wang、Xiangzhou Guo、Chengming Li、Xiaohai Xu 等。2021。Milvus：专用向量数据管理系统。载于《2021年国际数据管理会议论文集》。2614–2627。[53] Jingdong Wang、Ting Zhang、Nicu Sebe、Heng Tao Shen 等。2017。学习哈希综述。《IEEE模式分析与机器智能汇刊》40，4（2017），769–790。[54] Mengzhao Wang、Weizhi Xu、Xiaomeng Yi、Songlin Wu、Zhangyang Peng、Xiangyu Ke、Yunjun Gao、Xiaoliang Xu、Rentong Guo 和 Charles Xie。2024。Starling：用于数据段上高维向量相似性搜索的I/O高效磁盘驻留图索引框架。《ACM数据管理学报》2，1（2024），1–27。[55] Runhui Wang 和 Dong Deng。2020。DeltaPQ：用于高维相似性搜索的无损乘积量化编码压缩。《VLDB基金会学报》13，13（2020），3603–3616。[56] Xin-Jing Wang、Lei Zhang、Feng Jing 和 Wei-Ying Ma。2006。Annosearch：通过搜索实现图像自动标注。载于《2006年IEEE计算机学会计算机视觉与模式识别会议（CVPR'06）》，第2卷。IEEE，1483–1490。[57] Roger Weber、Hans-Jörg Schek 和 Stephen Blott。1998。高维空间相似性搜索方法的定量分析与性能研究。载于《VLDB》，第98卷。194–205。[58] Roger Weber、Hans-Jörg Schek 和 Stephen Blott。1998。高维空间相似性搜索方法的定量分析与性能研究。载于《VLDB》，第98卷。194–205。[59] Chuangxian Wei、Bin Wu、Sheng Wang、Renjie Lou、Chaoqun Zhan、Feifei Li 和 Yuanzhe Cai。2020。AnalyticDB-V：面向结构化与非结构化数据查询融合的混合分析引擎。《VLDB基金会学报》13，12（2020），3152–3165。[60] Yair Weiss、Antonio Torralba 和 Rob Fergus。2008。谱哈希。《神经信息处理系统进展》21（2008）。[61] Yuhuai Wu、Markus N Rabe、DeLesley Hutchins 和 Christian Szegedy。2022。记忆型Transformer。arXiv预印本 arXiv:2203.08913（2022）。

[62] Yuming Xu、Hengyu Liang、Jin Li、Shuotao Xu、Qi Chen、Qianxi Zhang、Cheng Li、Ziyue Yang、Fan Yang 等。2023。SPFresh：十亿规模向量搜索的增量原地更新。载于第*29*届操作系统原理研讨会论文集。545–561。[63] Wen Yang、Tao Li、Gai Fang 和 Hong Wei。2020。Pase：用于近似最近邻搜索的 PostgreSQL 超高维扩展。载于*2020*年*ACM SIGMOD*数据管理国际会议论文集。2241–2253。[64] Peter N Yianilos。1993。通用度量空间中最近邻搜索的数据结构与算法。载于第四届*ACM-SIAM*离散算法研讨会论文集。311–321。[65] Jialiang Zhang、Jing Li 和 Soroosh Khoram。2018。基于 OpenCL FPGA 的高效大规模近似最近邻搜索。载于*2018*年*IEEE/CVF*计算机视觉与模式识别会议。IEEE，4924–4932。[66] Jin Zhang、Defu Lian、Haodi Zhang、Baoyun Wang 和 Enhong Chen。2023。最大内积搜索的查询感知量化。载于*AAAI*人工智能会议论文集，第37卷。4875–4883。[67] Qianxi Zhang、Shuotao Xu、Qi Chen、Guoxin Sui、Jiadong Xie、Zhizhen Cai、Yaoqi Chen、Yinxuan He、Yuqing Yang、Fan Yang 等。2023。{VBASE}：通过松弛单调性统一在线向量相似性搜索与关系查询。载于第*17*届*USENIX*操作系统设计与实现研讨会（*OSDI 23*）。377–395。[68] Weijie Zhao、Shulong Tan 和 Ping Li。2020。Song：GPU上的近似最近邻搜索。载于*2020*年*IEEE*第*36*届国际数据工程会议（*ICDE*）。IEEE，1033–1044。[69] Zilliztech。2023。*VectorDBBench*：向量数据库基准测试工具。https://github. com/zilliztech/VectorDBBench

定理 4.1 的证明

证明。令 Q_l 为在以 x 为中心、半径为 l 的超球体上均匀分布的查询集合。

我们可以将期望展开如下：

$$L(c', c, Q_l) = E_{||R||=l}[ReLU(-\cos\alpha) \cdot (||q-c'||^2 - ||q-x||^2)]$$
$$= \int_{\frac{\pi}{2}}^{\pi} -\cos\alpha \, E_{||R||=l}[(R-r')^2 - R^2 \mid \frac{\langle R, r\rangle}{||R|| \cdot ||r||} = \alpha]$$
$$dP(\frac{\langle R, r\rangle}{||R|| \cdot ||r||} < \alpha)$$

为了计算这个积分，我们将 R 和 r' 分解为平行于 r 和与 r 正交的分量。如图 4 所示，我们分别将这些分量称为 $R_\parallel$、$R_\perp$、$r'_\parallel$、$r'_\perp$。此外，我们令 $\theta = \angle cxc'$、β 为 $R_\perp$ 与 $r_\perp$ 之间的角度。然后，在给定 r、r'、θ 和 α 的情况下，简化这个内层期望：

$$E_{||R||=l}[(R-r')^2 - R^2 \mid \frac{\langle R, r\rangle}{||R|| \cdot ||r||} = \alpha]$$
$$= E_{||R||=l}[(R_\parallel - r'_\parallel)^2 + (R_\perp - r'_\perp)^2 - R_\parallel^2 - R_\perp^2 \mid ||R_\parallel|| = l\cos\alpha]$$
$$= E_{||R||=l}[r'^2_\parallel - 2R_\parallel r'_\parallel) + r'^2_\perp - 2r'_\perp R_\perp \mid ||R_\parallel|| = l\cos\alpha]$$
$$= E_{||R||=l}[||r'||\cos\theta(||r'||\cos\theta - 2l\cos\alpha) + ||r'||^2\sin^2\theta - 2r'_\perp R_\perp \mid ||R_\parallel|| = l\cos\alpha]$$
$$= E_\beta[||r'||^2 - 2l||r'||\cos\theta\cos\alpha - 2l||r'||\sin\theta\sin\alpha\cos\beta]$$
$$= ||r'|| \cdot (||r'|| - 2l\cos\theta\cos\alpha - 2l\sin\theta\sin\alpha E[\cos\beta])$$
$$= ||r'|| \cdot (||r'|| - 2l\cos\theta\cos\alpha)$$

同时，$dP(\frac{\langle R, r\rangle}{||R|| \cdot ||r||} < \alpha)$ 与半径为 $l\sin\alpha$ 的 $(D-1)$ 维超球体的表面积成正比。因此，对于某个常数 A，我们将其表示为 $A\sin^{D-2}\alpha$。于是积分变为

$$L(c', c, Q_l) = \int_{\frac{\pi}{2}}^{\pi} -\cos\alpha||r'||(||r'|| - 2l\cos\theta\cos\alpha)A\sin^{D-2}\alpha d\alpha$$
$$= -A||r'||^2 \int_{\frac{\pi}{2}}^{\pi} \cos\alpha\sin^{D-2}\alpha d\alpha + 2Al||r'||\cos\theta\int_{\frac{\pi}{2}}^{\pi}\cos^2\alpha\sin^{D-2}\alpha d\alpha$$
$$= \frac{A||r'||^2}{D-1} + Al||r'||\cos\theta\int_0^{\pi}(\sin^{D-2}\alpha - \sin^D\alpha)d\alpha$$

现在我们定义 $I_D = \int_0^{\pi}\sin^D\alpha d\alpha$。

$$I_D = 2\int_0^{\pi/2}\sin^D\alpha d\alpha$$
$$= -2\cos\alpha\sin^{D-1}\alpha\Big|_0^{\pi/2} - 2\int_0^{\pi/2}\cos\alpha(-(D-1)\cos\alpha\sin^{D-2}\alpha)d\alpha$$
$$= 2(D-1)\int_0^{\pi/2}\cos^2\alpha\sin^{D-2}\alpha d\alpha$$
$$= (D-1)I_{D-2} - (D-1)I_D$$

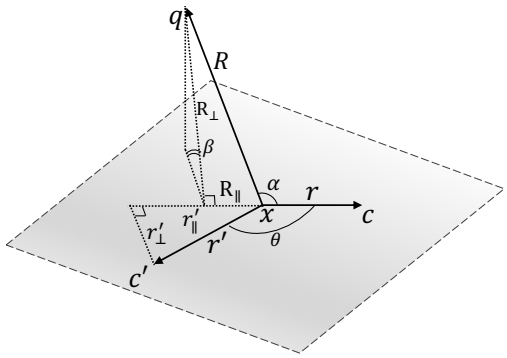


图18：向量的详细几何关系。

合并各项，可得 $I_D = \frac{D-1}{D} I_{D-2}$。然后，损失满足：

$$L(c', c, Q_l) = \frac{A||r'||^2}{D-1} + Al||r'||\cos\theta(I_{D-2} - I_D)$$
$$= \frac{A||r'||^2}{D-1} + Al||r'||\cos\theta\frac{I_D}{D-1}$$
$$= \frac{A}{D-1}(||r'||^2 + I_D l\cos\theta||r'||)$$
$$= \frac{A}{D-1}(||r'||^2 + \frac{I_D l}{||r||}r^T r')$$

注意，超球体 Q 由许多超球面表面 Q_l 组成。我们求解 $L(c', c, Q)$：

$$L(c', c, Q) = \int_0^{l_m} L(c', c, Q_l)\cdot\frac{Dl^{D-1}}{l_m^D}dl$$
$$= \int_0^{l_m}\frac{A}{D-1}(||r'||^2 + \frac{I_D l}{||r||}r^T r')\frac{Dl^{D-1}}{l_m^D}dl$$
$$= \frac{A}{D-1}(||r'||^2 + \int_0^{l_m}\frac{DI_D l^D}{l_m^D||r||}r^T r' dl)$$
$$= \frac{A}{D-1}(||r'||^2 + \frac{DI_D l_m}{(D+1)||r||}r^T r')$$
$$\propto ||r'||^2 + \lambda r^T r'$$

where $\lambda = \frac{DI_D l_m}{(D+1)||r||} > 0$.

□