软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(3):1058−1083 [doi: 10.13328/j.cnki.jos.007513] [CSTR: 32375.14.jos.007513]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

LSMDiskANN: 更新友好型磁盘向量索引框架*

邱海浪[1], 彭煜玮[1], 彭智勇[1,2]

[1](武汉大学 计算机学院, 湖北 武汉 430072)
[2](武汉大学 大数据研究院, 湖北 武汉 430072)
通信作者: 彭煜玮, E-mail: ywpeng@whu.edu.cn

摘　要: 在大模型时代, 向量数据库的广泛应用推动了向量索引规模的急剧膨胀. 如何在磁盘级向量索引中高效支持大规模向量的更新操作, 并同时提供高性能的查询服务, 已成为近年来的重要研究课题. 针对当前领先算法 FreshDiskANN 在查询与更新混合负载场景中面临的查询吞吐瓶颈和极端查询延迟过高等问题, 受到日志合并思想在次级索引中的成功应用启发, 提出了一种基于 LSM (log-structured merge) 思想的更新友好型磁盘向量索引框架 LSMDiskANN. 在继承 FreshDiskANN 架构的基础上, 设计并实现了包含磁盘中间层的 3 层架构, 同时引入了磁盘组件搜索参数的动态确定机制以及面向合并操作删除阶段的重布局算法, 从而进一步降低查询延迟和合并过程中的 I/O 开销. 实验结果表明, 在多个经典大规模高维向量数据集上, LSMDiskANN 系统查询吞吐量最高提升 35.5%, 更新吞吐量最高提升 14.24%, 极端查询延迟最多降低 73.45%, 所提出框架和策略能够有效提升系统在混合负载场景下的整体性能与稳定性.

关键词: 向量数据库; 磁盘向量索引; 动态向量索引; 日志合并
中图法分类号: TP311

中文引用格式: 邱海浪, 彭煜玮, 彭智勇. LSMDiskANN: 更新友好型磁盘向量索引框架. 软件学报, 2026, 37(3): 1058–1083. http://www.jos.org.cn/1000-9825/7513.htm

英文引用格式: Qiu HL, Peng YW, Peng ZY. LSMDiskANN: Update-friendly Disk-resident Vector Index Framework. Ruan Jian Xue Bao/Journal of Software, 2026, 37(3): 1058–1083 (in Chinese). http://www.jos.org.cn/1000-9825/7513.htm

LSMDiskANN: Update-friendly Disk-resident Vector Index Framework

QIU Hai-Lang[1], PENG Yu-Wei[1], PENG Zhi-Yong[1,2]

[1](School of Computer Science, Wuhan University, Wuhan 430072, China)
[2](Big Data Institute, Wuhan University, Wuhan 430072, China)

Abstract: In the era of large models, the widespread use of vector databases has led to a rapid expansion in the scale of vector indexes. How to efficiently support large-scale vector updates in disk-based vector indexes while maintaining high query performance has become an important research problem in recent years. FreshDiskANN, as a leading algorithm, suffers from query throughput bottlenecks and high tail latency under mixed query-update workloads. Inspired by the successful application of log-structured merge (LSM) in secondary indexes, LSMDiskANN is proposed as an update-friendly disk-resident vector index framework based on the LSM paradigm. Building on the FreshDiskANN architecture, a three-level structure including a disk intermediate level is designed and implemented. In addition, a dynamic parameter selection mechanism for disk component search and a re-layout strategy for the deletion phase of compaction are introduced to further reduce query latency and I/O overhead during merges. Experimental results show that on multiple large-scale, high-dimensional datasets, query throughput is improved by up to 35.5%, update throughput by up to 14.24%, and tail query latency is reduced by up to 73.45%. The proposed framework and strategies effectively enhance overall performance and stability under mixed workloads.

* 基金项目: 国家重点研发计划 (2023YFB4503604)
本文由“向量数据库及 DB4LLM 技术”专题特约编辑高宏教授、李国良教授、张蓉教授推荐.
收稿时间: 2025-05-06; 修改时间: 2025-06-30, 2025-08-14; 采用时间: 2025-08-20; jos 在线出版时间: 2025-09-02
CNKI 网络首发时间: 2026-01-15

Key words: vector database; disk vector index; dynamic vector index; log-structured merge (LSM)

在“万物皆向量”的时代, 向量数据库应运而生. 各类模态的数据 (如文本、图像、视频等) 可以通过自然语言处理、图像处理等深度学习模型转换为高维嵌入向量, 进而被深度学习模型利用, 实现多模态检索功能[1–3]. 其中, 最基本且核心的查询操作是 K 近邻 (k-nearest neighbor, KNN) 查询. 向量数据库则用于存储和索引这些高维向量并提供高效的 KNN 查询服务. 但是, 由于众所周知的“高维诅咒”问题[4], 在可接受的时间内获得精确的 KNN 结果几乎是不可能的. 因此, 在向量数据库的应用场景中, KNN 查询通常指的是 K 近似近邻 (K-approximate nearest neighbor, KANN) 查询. 过去 20 年中, 为提升向量查询效率, 研究者提出了多种向量索引技术, 主要可分为两类: 基于图的索引[5–8]与基于分区的索引[9–12]. 其中图索引由于其优越的搜索性能而备受青睐, 因此也成为本文研究的重点方向.

然而, 目前主流的图索引方法大多基于内存构建与更新. 随着数据量持续暴涨, 仅依赖内存显然无法满足大规模向量数据集的需求. 因此, 近年来出现了大量基于磁盘的向量索引研究[11,13,14], 这些研究通过向量压缩算法与导航结构设计, 以较小的搜索性能损失换取低内存占用, 并控制磁盘 I/O 访问量, 从而实现高效的 KNN 搜索.

在数据量急剧增长的同时, 另一个核心挑战随之出现: 如何支持大规模向量更新. 如图 1 所示, 在典型的检索增强生成 (retrieval-augmented generation, RAG) 场景中, 向量数据库背后的知识库需要持续动态更新, 这些更新可能来自视频网站每日新增的视频数据[15]或是学术文献数据库的日常扩充[16]等. 围绕这方面的研究主要从以下两点出发: (1) 动态向量索引算法的设计[17–19], 传统索引往往采用周期性重建的方式应对更新, 这类工作尝试在现有索引结构上引入增量更新机制, 从而避免全量重建带来的高开销; (2) 面向高更新吞吐量的索引架构优化[18,20,21], 这些工作则聚焦于调整索引布局与层次结构, 进而提高索引系统对高频写入与查询的支持能力.

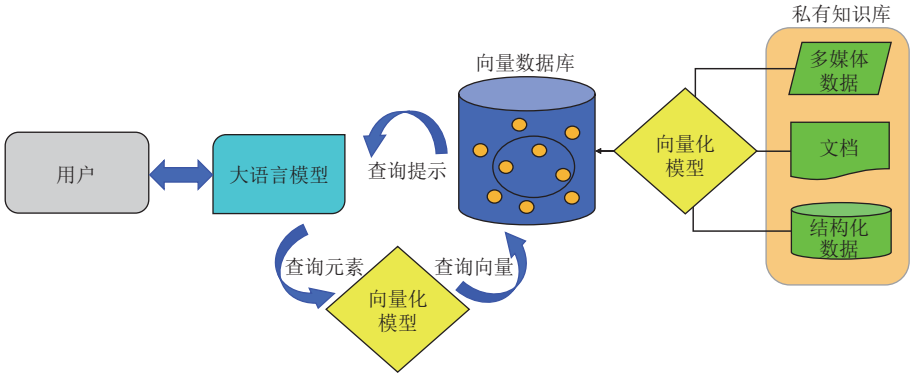


图 1 RAG 场景示例

因此, 目前大部分专用向量数据库均建立于 NoSQL 数据库之上, 比如 Milvus[21]、Qdrant[22]和 Weaviate[23]. 通过 LSM (log-structured merge) 树架构, 这些专用向量数据库实现了数据表的高写入吞吐、低内存占用与高效主键点查找. 然而, 在这种架构中, 向量索引作为向量列的次级索引, 如何在不影响主表写入性能的前提下高效更新, 是一个需要解决的关键问题.

为此, FreshDiskANN[18]被提出, 它是目前最具代表性的图结构磁盘向量索引方法之一, 也是 BIGANN-2023 比赛 Stream Search 赛道的基准方法[24]. FreshDiskANN 的架构类似单层 LSM 树, 采用 FreshVamana 算法, 在内存中进行增量图构建, 并周期性地将其合并到磁盘索引中. 在向量搜索与更新问题上, 该算法展现了优秀的搜索与更新性能. 然而, 我们在实验中观察到其仍存在以下不足. (1) 合并操作频率高: 每当内存索引写满时都会触发一次与磁盘索引的合并, 造成过高的合并频率, 严重影响前台查询性能. (2) 磁盘 I/O 利用率低: 每次合并需要遍历磁盘索引中所有向量点, 但根据 Greator[25]的分析, 实际需要更新的点可能仅占 4%, 导致大量不必要的 I/O. (3) 合并操作中删除阶段的磁盘 I/O 开销大: 整个合并过程的删除阶段需顺序扫描磁盘索引两趟, 带来额外的磁盘 I/O 开销.

针对上述问题, 本文提出了 LSMDiskANN, 一个以 FreshDiskANN 为基础的更新友好型磁盘向量索引框架. 为

了降低频繁合并造成的系统性能影响, 本文基于 FreshDiskANN 的架构和索引算法, 将其拓展为内存层、磁盘中间层和磁盘基本层 3 层架构; 为了降低冗余查询的额外开销, 本文提出了搜索参数动态确定机制, 在层间以及层内采用不同搜索参数; 为了减少合并操作删除阶段的 I/O 开销, 本文提出了磁盘重布局算法.

综上, 本文的主要贡献包括以下 3 点.

(1) 基于 LSM 思想的更新友好型磁盘向量索引框架: 在 FreshDiskANN 架构基础上, 引入磁盘中间层与刷新操作, 有效降低合并操作频率, 同时将磁盘中间层作为缓冲区域, 提升了每次合并操作的数据量, 从而增加其中被更新点的占比, 最终提高 I/O 利用率. 实验结果表明, 该方法最高能够增加 35.5% 查询吞吐量和 14.2% 更新吞吐量.

(2) 搜索参数动态确定机制: 在引入磁盘中间层后, 多层索引结构会带来查询路径冗余问题, 导致查询延迟增加. 基于“所包含向量数据越多的磁盘索引包含最终结果的可能性越高”的启发式假设, 本文设计了层间动态搜索参数策略. 进一步, 结合 DiskANN 中磁盘索引的量化信息反映索引数据分布状况的特点, 本文定义了查询点与磁盘索引的距离计算方法, 并基于该距离设计了层内动态搜索参数策略. 在查询时采用多种策略相结合以显著降低查询延迟. 实验结果表明, 该机制平均提升查询 QPS (query per second) 达 65.86%.

(3) 面向合并操作删除阶段的重布局算法: 本文借鉴面向查询操作的向量索引磁盘重布局算法[26], 并结合合并操作删除阶段顺序读取特点, 优化向量索引点在磁盘上的物理排序, 使得原本需要二次顺序扫描的操作转化为“一次顺序扫描+少量随机读取”, 最终大幅减少合并操作删除阶段的 I/O 开销. 实验结果显示, 该方法平均可降低删除阶段 41.36% 的 I/O 开销.

本文第 1 节介绍磁盘动态向量索引与 LSM 次级索引相关的研究背景. 第 2 节介绍所需的基础知识, 包括相关概念的定义、量化压缩与 FreshDiskANN 架构等. 第 3 节介绍本文提出的基于 LSM 思想的更新友好型磁盘向量索引框架及相关优化. 第 4 节通过对比实验与消融分析验证所提框架的有效性. 第 5 节总结全文.

1 相关工作

1.1 向量索引算法

K 近似近邻搜索 (KANN) 是数据挖掘领域的经典问题, 近年来随着以检索增强生成 (RAG) 为代表的高维向量检索需求的兴起, KANN 再次成为研究热点. 该领域近年已有多篇综述和基准测试研究[26−30], 为理解该领域发展脉络提供了良好基础. 从技术角度看, 向量索引算法主要分为两大类: (1) 基于图的算法: 将每个向量抽象为图中的一个节点, 依据特定规则构建索引图[5,7,8,31], 查询阶段则借助剪枝策略[6,32,33]与图遍历方法[34]完成高效搜索; (2) 基于分区的算法: 可进一步细分为基于树索引的算法[12,35]、基于哈希索引的算法[10,36,37]和基于倒排索引的算法[9,11,38], 其核心关注以下 3 个问题: (1) 如何划分分区; (2) 查询时需访问哪些分区; (3) 如何在分区内执行高效搜索, 由此衍生出分区策略、导航策略与执行策略这 3 条研究路径.

1.2 磁盘向量索引

随着自然语言处理等领域中向量表示技术的发展, 为了保留更多原始模态的特征, 向量维度呈现数量级增长, 从传统的几十维增长至如今常见的数百甚至上千维, 如 DEEP[39]中的 96 维、GIST[9]中的 960 维、WIT-Image[40]中的 2 048 维. 同时在现今互联网与物联网加持下, 向量数据规模也在呈现数量级增长, 比如 YouTube8M[15]、SIFT1B[9]、Microsoft SPACEV-1B[41]等更多大规模向量数据集. 多种因素影响下, 向量存储压力也随之倍增.

传统向量索引多为纯内存索引, 其假设索引结构和向量数据完全驻留在内存中, 但这一前提在大规模数据场景下显然不现实. 因此, 近年来大量研究转向基于磁盘的向量索引技术[11,13,14,42,43], 其中最具代表性的则是微软提出的 SPANN[11]和 DiskANN[13]: (1) SPANN 基于倒排索引, 将倒排列表持久化在磁盘上, 将中心点构建成内存导航图, 并保证在少量倒排列表遍历下即可完成查询, 从而在保证查询准确性、查询延迟的同时显著节省内存开销. 其结构简单, 构建速度快, 但为保证高召回率需进行数据冗余存储, 磁盘开销较大; (2) DiskANN 基于图索引, 将原始向量及邻接表存于磁盘, 仅将量化压缩编码与码表加载到内存中. 在查询时, 通过压缩码进行距离估算, 再结合贪婪遍历策略逐步读取邻居信息与原始向量, 最终完成精排查询. 得益于图结构的高收敛性, 每次查询需访问的节点

数量有限, 从而实现内存占用与查询效率的平衡.

1.3 动态向量索引

伴随向量数据体量的迅猛增长, 其更新频率也呈现爆炸式提升. 例如, YouTube 每分钟有超过 500 h 的视频内容上传[44], 京东每天有亿级的图像数据产生[45], 阿里巴巴在双十一期间产生超 500 PB 非结构化数据[20]. 对于传统向量索引, 处理更新的策略就是定期重建整个索引, 这在实际生产环境中成本极高, 种种需求催生了动态向量索引的研究.

动态向量索引的研究方向主要包括两方面. (1) 一方面是改进现有索引算法, 使其支持更新操作, 从而避免频繁重建整个索引. 这方面的两个代表性工作是微软提出的 SPFresh[17]与 FreshDiskANN[18], 二者分别基于 SPANN 和 DiskANN 拓展了更新能力. 前者解决了倒排索引在更新过程中倒排列表长度失衡问题; 后者解决了图索引更新效率等问题. (2) 另一方面则聚焦于设计更适合高频更新场景的索引架构, 进而提升索引的吞吐量. 例如: AnalyticDB-V[20]通过批处理层和流处理层的两层架构设计, 实现数据的对外无感更新, 但其底层向量索引仍是静态, 从而引发更新性能问题; 作为典型 NoSQL 数据库, Milvus[21]采用基于 LSM 的存储架构, 将数据表划分为多个分片, 并在每个分片上构建静态向量索引, 在数据表进行合并时同步重建向量索引, 达到无感更新效果. 但是受限于嵌入式索引与数据表高度耦合的缺点; FreshDiskANN 本质上可看作单层 LSM 结构, 其在内存中维护增量数据并定期合并至磁盘主索引. 但是由于单层的限制, 每当内存索引写满时即需合并, 导致合并操作过于频繁, 严重影响前台查询性能.

此外, 近期还有两项针对 FreshDiskANN 的改进研究, IP-DiskANN[46]为减少合并操作删除阶段的邻居修补开销, 通过查询结果来近似删除点的入边邻居列表, 但该方法会不断累积逻辑删除的冗余边, 从而需要额外引入索引的定期清理操作; Greator[25]则引入图拓扑结构独立存储方案, 将图拓扑结构作为一种向量索引的索引, 结合局部更新策略和近似剪枝策略提高小批量更新场景下的合并操作效率. 但是, 小批量更新场景的实际应用意义值得商榷, 且图拓扑结构的额外单独存储在大部分高维向量数据集下, 其所引发的额外存储开销比例不容小觑 (如 SIFT 中, 在邻居列表长度为 64 时, 比例为 50%). 总而言之, 上述两项研究均侧重于算法层面的调整与修改, 而本文的创新则侧重于调整索引布局与架构设计, 因此彼此互补且不冲突.

1.4 LSM 次级索引

在 NoSQL 数据库中, 为支持大规模数据的高效写入, 同时兼具灵活性与可拓展性, 基于 LSM 的存储架构被广泛采用. 其将主表存储设计为多层可合并的组件, 兼顾写入吞吐与查询性能. 但实际应用中, 除了主键查询, 往往还需在非主键列上建立索引, 即“次级索引”. 如何保证次级索引在高写入负载下仍能高效更新, 成为 NoSQL 领域的研究焦点[47]. 其中, 一大解决方案则是将次级索引也 LSM 化, 这类索引统称为 LSM 次级索引, 包括: LSM 倒排索引[48]、LSM B 树[49]、LSM R 树[49−51]等, 其中, 本文所提框架设计灵感则来源于 LSM R 树——用于支持空间数据的 KNN 与范围查询的树索引结构, 在 Apache AxterixDB[49]中有完整的开源实现.

在架构上, LSM R 树由内存层与磁盘层组成, 称为“组件”. 内存层组件包含 R 树与记录删除键的 B+树, 磁盘层组件则额外加入布隆过滤器用于快速过滤删除标记. 当写入操作执行时, 数据首先写入内存组件, 内存组件写满后刷新为磁盘组件. 当磁盘组件数量超过阈值 C 时, 将根据合并策略 (如 Constant 或 Prefix) 执行合并. 当 KNN 搜索执行时, 需遍历所有组件, 按照生命周期从老到新逐个进行查询与删除键的过滤. 通过 LSM 化改造, AsterixDB 显著提升了读写混合场景下的系统吞吐与 QPS, 也为本文提出的基于 LSM 思想的磁盘向量索引框架提供了强有力的可行性支撑.

2 基础知识

本文所提架构主要基于 FreshDiskANN[18]进行改进, 下面就相关概念和系统基本过程予以介绍.

2.1 K 近似近邻查询

K 近邻 (KNN) 查询常用于从大规模数据集中检索与给定查询项相似的数据. 在向量数据库中, 数据以向量形式表示, KNN 查询即为寻找与查询向量最接近的 K 个向量. 在实际应用中, KNN 查询广泛用于如下场景: (1) 检索

增强生成 (RAG): 在大模型提示工程中, 对输入提示中提及的实体进行向量搜索, 返回相关实体向量作为提示的一部分, 以增强大模型的上下文知识, 减少幻觉生成; (2) 多模态检索与推荐: 用户输入如图像等多模态数据后, 系统检索相似向量 (如图像向量) 并返回相应内容, 用于推荐与相关性排序.

在实践与理论研究中, 已广泛证明“高维诅咒”现象是向量搜索中不可忽视的难题: 在高维空间中, 精确地获取 K 近邻的唯一可行方式往往是遍历全集, 而这种代价在真实系统中难以接受. 为此, 研究与工程中普遍采用 K 近似近邻 (KANN) 查询, 对精度要求进行松弛, 以换取查询效率. 本文给出 K 近邻查询定义.

定义 1 (K 近邻查询). 设有元素集合 S 和预定义距离函数 D, 给定查询结果集大小 k 的情况下, 输入一个查询元素 q, K 近邻查询目标是找到 $S' \subseteq S$, $|S'| = k$, s.t. $\forall x' \in S'$, $\forall x \in S \backslash S'$, $D(q, x') \leqslant D(q, x)$.

而 K 近似近邻查询旨在返回尽可能接近真实 K 近邻集合的结果. 其查询精度通常通过“召回率”指标衡量. 召回率越高, 则认为 K 近似近邻查询结果越精确, 本文给出 召回率@k 定义如下.

定义 2 (召回率@k). 设有元素集合 S, 给定一个查询元素 q, 假设 $G \subseteq S, |G| = k$ 且是 q 在 S 中的真实 K 近邻集合, S' 是查询的输出集合, 则有如下 召回率@k 定义公式:

$$召回率@k = \frac{|S' \cap G|}{k} \tag{1}$$

定义 3 (近似 K 近邻查询). 设有元素集合 S 和预定义距离函数 D, 给定查询结果集大小 k 的情况下, 输入一个查询元素 q, 输出查询集合 $S' \subseteq S$ 使得 召回率@k 最大化.

本文进一步关注更新场景下的 K 近似近邻查询性能, 沿用 FreshKANNS (fresh K-approximate nearest neighbor search) 定义[18], 给出如下形式化定义.

定义 4 (FreshKANNS)[18]. 设有随时间演化的元素集合 P (在时间 t 具有状态 P_t), 目标是在当前数据状态 P_t 上构建并维护一个动态索引, 该索引支持以下 3 类操作: (1) 插入新元素; (2) 删除已有元素; (3) 对于查询元素 q, 在当前数据状态 P_t 上执行 K 近似近邻搜索.

2.2 量化压缩

为了减小内存开销并提升距离计算效率, 图结构的磁盘向量索引通常对原始向量进行压缩, 其中量化压缩[9,38]是一种常见的压缩方法, 该方法将压缩码与码表加载到内存中, 在查询时以近似方式估算向量间距离.

如图 2 所示, 量化压缩主要流程如下.

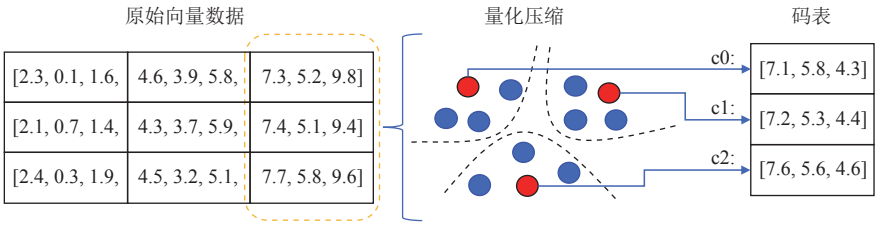


图 2 量化压缩示意图

(1) 分块处理: 将高维向量划分为多个低维子向量块.

(2) K-means 量化: 对每个子块使用 K-means 聚类, 生成对应的聚类中心 (即码表).

(3) 编码表示: 原始子向量以所属簇的中心索引表示, 即构成压缩码.

(4) 码表组合: 多个子块的聚类中心集合经笛卡尔积组合后, 可近似恢复原始向量空间.

简言之, 量化将连续向量空间离散化, 使得任一原始向量可由若干聚类中心近似重构, 从而达到有损压缩目的. 其核心思想是在每个子向量块中选取代表性点, 再将代表点通过维度组合表示整个高维空间.

在查询阶段, 如图 3 所示, 系统采用 ADC (asymmetric distance computation) 策略, 使用原始查询向量 (图中星状点) 与向量码表中聚类中心 (图中红点) 间的距离进行近似计算. 由于查询向量是未压缩的, 只需事先计算其与所有聚类中心的距离, 便可通过查表方式高效获取压缩向量的近似距离, 大幅提升计算速度.

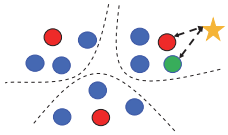


图 3　ADC 策略示意图

2.3　FreshDiskANN

FreshDiskANN 作为基于图的动态磁盘向量索引中的代表性算法, 因其在查询效率与更新能力之间实现良好平衡, 成为本文的重要对比对象. 以下对其核心架构与关键机制进行简要介绍.

2.3.1　架构概述

FreshDiskANN 采用内存层与磁盘层共同组成的双层结构, 我们将两层中的基本结构单元统称为组件.

(1) 内存层组件: 分为两类: ① 读写组件, 支持插入、删除与查询操作; ② 只读组件, 仅支持查询操作. 二者结构相同, 均包含: ① 一个删除键集合 (记录逻辑删除的向量 ID); ② 一个基于内存构建的 Vamana 图索引. 系统运行过程中, 内存层至多存在一个读写组件, 其余均为只读组件.

(2) 磁盘层组件: 单一的长期数据组件, 使用 DiskANN 索引结构, 并作为系统的持久存储层. 通常, 系统初始化时会加载一批初始向量构建磁盘层索引, 具体过程包括先在内存中构建 Vamana 图, 随后将其序列化为 DiskANN 磁盘格式. 同时, 对所有向量进行量化压缩, 生成压缩码与码表并保存在内存中, 以支持后续的高效距离计算.

2.3.2　核心操作机制

(1) 插入/删除操作: 当插入/删除一个点时, 都是在内存层中的读写组件进行. 新向量插入时, 首先在读写组件上执行一次近似 KNN 查询, 获取其初始邻接出边. 随后, 将新点与出边一起插入 Vamana 索引中, 并尝试将每条出边对应的反向边加入目标点的邻接入边集合, 同时根据 Vamana 剪枝策略淘汰部分节点; 删除时, 则采用懒删除策略, 即不立即从索引中物理删除点, 而是将其主键记录到删除键集合中, 实际查询中再进行后过滤.

(2) 查询操作: 查询向量时, 系统会在内存层与磁盘层的所有组件依次执行相同参数的查询操作. 随后, 将多个结果集合并, 并根据内存中所有删除键集合过滤逻辑删除的向量, 最终返回过滤后的查询结果.

(3) 合并操作: 当读写组件的数据量超过预设最大阈值的一半时, 系统将其转化为只读组件, 并创建新的读写组件处理后续数据. 同时, 触发一次将该只读组件合并入磁盘层组件的操作. 如图 4 所示, 整个合并操作分为以下 3 个阶段.

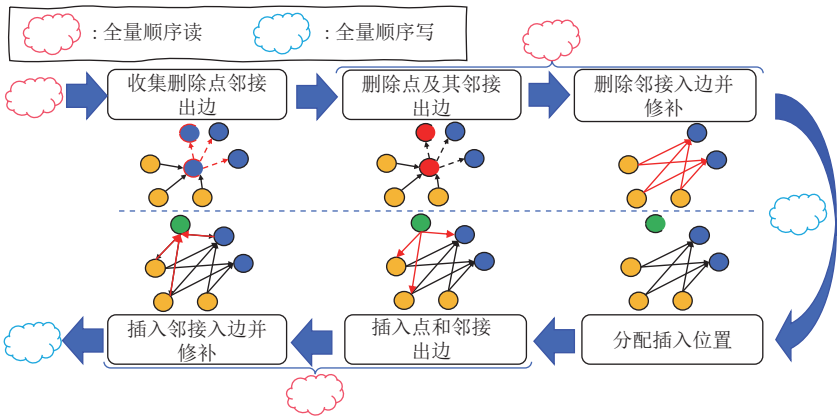


图 4　FreshDiskANN 合并操作示意图

● 删除阶段: 首先, 根据只读组件的删除键集合, 顺序扫描磁盘层长期数据组件索引, 收集所有被删除点的邻接表, 然后, 再重新顺序扫描磁盘层长期数据组件中的所有索引点. 如果是删除点, 则直接移除; 如果邻接表中包含删除点, 则将被删除点的出边合并入当前点的邻接表, 若合并后邻接表长度超出阈值, 则根据 Vamana 剪枝策略淘

汰部分节点.

● 插入阶段: 类似于插入操作, 对待插入的新点, 在磁盘层长期数据组件上进行一次 K 近邻查询以获取初始出边集合, 并将其与反向边一起暂存于 Δ 结构中, 以便修补阶段应用.

● 修补阶段: 顺序扫描磁盘层长期数据组件所有索引点, 如果是插入点, 将其与 Δ 结构中存储的出边一起插入至索引; 如果是已有点, 若其邻接出边集合因 Δ 结构中暂存边的插入导致长度超限, 则根据 Vamana 剪枝策略淘汰部分节点, 维持索引结构紧凑性.

总而言之, FreshDiskANN 通过上述双层索引结构与高效的更新/合并机制, 实现了在支持高吞吐插入的同时, 维持高质量的近似邻居查询性能. 其设计理念对后续系统架构具有重要参考意义.

3 基于 LSM 思想的更新友好磁盘向量索引框架

在实际测试中, 我们观察到 FreshDiskANN 的后台合并操作频率偏高, 频繁抢占搜索线程资源, 进而对前台查询性能造成负面影响. 为解决该问题, 本文提出了一种基于 LSM 思想的更新友好磁盘向量索引框架, 旨在通过牺牲部分索引静态搜索性能, 以降低合并频率、提高单次合并数据量, 从而提升整体查询吞吐能力.

该框架受 LSM R 树[49]启发, 在 FreshDiskANN 的架构基础上进行了关键改进. 具体而言, 本文引入了一个新的磁盘中间层, 并配合该层设计了新的操作逻辑. 此外, 为了缓解组件数量增加可能导致的查询冗余和合并操作删除阶段的 I/O 开销增长, 本文还提出了磁盘组件搜索参数动态调整机制与磁盘索引重布局策略.

3.1 架构设计

与 FreshDiskANN 双层架构不同, 本文提出的框架在内存层与磁盘层之间新增磁盘中间层. 如图 5 所示, 该结构从高层设计上看虽然较为简单, 但实际实现中涉及更复杂的组件管理逻辑. 从本文的实验结果来看, 这是一个思想上简单、实现上复杂、效果上有效的架构. 整个系统分为 3 层: 内存层、磁盘中间层、磁盘基本层, 每一层由一个或多个索引组件组成.

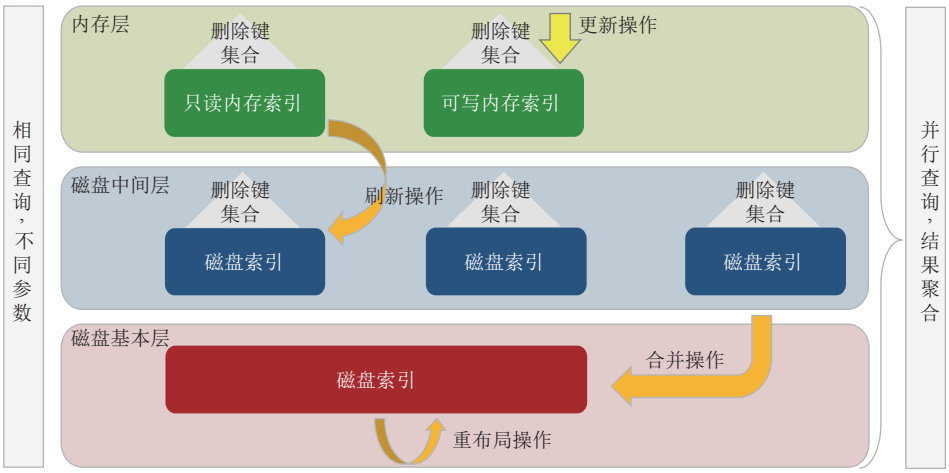


图 5 基于 LSM 思想的更新友好型磁盘向量索引框架图

3.1.1 架构概述

● 内存层组件: 与 FreshDiskANN 保持一致, 每个组件包含一个 Vamana 图索引和一个删除键集合. 整个内存层由若干个组件组成, 其中仅允许一个读写组件, 用来处理数据更新, 其他均为只读组件. 为限制内存开销, 通常控制每个组件的数据规模在较小范围内 (例如: 百万量级数据中限制每个组件不超过 32k 个向量).

● 磁盘中间层组件: 每个组件均由内存层组件转换生成, 结构上包含一个 DiskANN 索引与一个删除键集合. 整个磁盘中间层由若干个组件组成. 在其生命周期内, 这些组件不再接受修改, 属于只读、独立管理的结构单元.

● 磁盘基本层组件: 相当于 FreshDiskANN 的磁盘层, 包含一个长期存储的 DiskANN 索引. 该层组件承载系统的绝大多数数据, 生命周期最长. 合并时不做就地修改, 而是通过重建新组件并原子替换的方式完成结构更新, 确保查询操作不受影响.

3.1.2 插入/删除操作

插入与删除操作仅作用于内存层的读写组件, 具体流程遵循 FreshDiskANN 中的实现.

为了支持删除操作, 本文采取主动 (eager) 策略: 每个非磁盘基本层组件都维护一个删除键集合, 用于标记相对于其更老的组件中需要被逻辑删除的向量. 换句话说, 一个组件中实际可用的数据需排除所有更年轻组件中记录的删除键. 为减少查询时删除键集合的访问成本, 每一层还维护一个合并删除集合, 作为本层所有组件删除集合的并集, 用于快速过滤比当前层更老的层中的已删除向量.

3.1.3 查询操作

查询操作需要跨多个层级进行, 因此本文设计了多层索引查询算法 (见算法 1, 其中 L_{-1}、L_0、L_1 分别对应于内存层、磁盘中间层和磁盘基本层).

算法 1. 多层索引查询.

输入: 查询向量x_q, 返回结果集大小K, 层级索引集合$L = [L_{-1}, L_0, L_1]$;

输出: 最终结果集V.

1. **begin**
2. $R \leftarrow [\varnothing$ **for** i in $-1..1]$ //初始化中间结果集数组
3. $V \leftarrow \varnothing$ //初始化最终结果集V
4. $D \leftarrow [\varnothing$ **for** i in $-1..1]$ //初始化删除键集合
5. **for** $i \leftarrow 0$ to 1 //按照生命周期从新到老构建删除键集合
6. $D_i \leftarrow D_{i-1} + L_{i-1}.GetDeleteSet()$
7. **endfor**
8. **foreach** L_i in L
9. $R_i \leftarrow KNNQuery(L_i, x_q, D_i)$ //对各层分别进行K近邻查询
10. **endfor**
11. **for** $i \leftarrow -1$ to 1 // 将查询结果进行聚合
12. $V \leftarrow V \cup R_i$
13. **endfor**
14. $V \leftarrow$ sort V by vector distance ASC with limit K //对结果集V按照距离进行排序并保留前K个
15. **end**

由于不同层级的组件之间数据互不重叠, 为保证数据覆盖的完整性, 查询需在所有层中分别执行, 然后合并结果并做距离排序. 查询按算法 1 流程执行. 首先初始化各层中间结果集并准备好各层级索引查询所需要排除的删除键集合 (第 2–7 行), 然后再对各层级索引分别进行 K 近邻查询 (第 8–10 行), 再把过滤后的结果进行聚合 (第 11–13 行), 最后再将聚合结果按距离进行重排序并保留最近的 K 个 (第 14 行).

磁盘中间层的引入虽然提升了更新友好性, 但也带来了一个挑战: 组件数量增加导致的查询冗余. 本文提出以下 3 种配合使用的策略以有效缓解此问题.

(1) 并行查询策略: 在 FreshDiskANN 中, 其由若干个内存层组件和单个磁盘层组件组成, 其中内存层组件在搜索时因为不涉及磁盘 I/O, 所以其查询时间相比磁盘层组件搜索可以忽略不计, 采用串行查询即可满足性能需求. 然而本文框架中磁盘组件数量增加, 且涉及磁盘 I/O, 因此均可以采用并行查询策略, 有效降低查询延迟.

(2) 层间动态搜索参数策略: FreshDiskANN 以统一搜索列表长度 L_s 对所有组件进行查询, 由于图索引本身特

点, 其中直接影响搜索延迟的是搜索列表长度 L_s 而不是组件数据量, L_s 表示当前组件查询返回的结果集合长度, 为了保证高召回率, L_s 的设置通常远大于查询指定的 K, 从而导致冗余查询现象严重. 以 FreshDiskANN 中的默认值 $L_s = 75$, $K = 5$ 为例, 假如现有 1 个内存层组件和 1 个磁盘层组件, 则会查询得到长度为 150 的结果集合, 并进行重排序得到前 5 条结果. 如果本文照搬这个策略, 冗余查询带来的查询延迟将不可估量. 如第 3.1.1 节所言, 不同层组件的数据量大小差异很大. 因此, 直观地想, 数据量越大的组件包含最终 5 条结果的概率越大, 反之越小. 基于这种启发式思想, 本文在磁盘中间层组件与磁盘基本层组件采用不同的搜索参数. 假如内存层和磁盘基本层遵循 FreshDiskANN 的设置, 磁盘中间层设有 5 个组件且查询参数 $L_s = 15$, 那么其结果集长度则为 15×5+150=225, 因此即使是串行搜索, 其搜索时间也仅比 FreshDiskANN 高出 50%, 从而大大减少冗余查询.

(3) 层内动态搜索参数策略: 从理论上讲, 只要 $L_s \geqslant K$, 便存在获得 100% 召回率的可能. 基于此, 本文进一步提出了一种动态确定搜索列表长度的机制 (详见后文第 3.2 节), 可根据组件数量与查询特征自动调整 L_s, 最大程度减少冗余计算, 提高查询效率.

3.1.4 刷新操作

在引入磁盘中间层后, 系统需设计刷新机制以实现内存层组件向磁盘中间层组件的转换. 当内存层组件的数据量超过设定阈值时, 系统将触发一次刷新操作, 将其转化为磁盘中间层组件. 图 6 展示了 Vamana 索引在内存与磁盘中的两种数据布局形式的差异: (1) 内存形式下向量数据与邻接表分离存储, 而在磁盘形式中, 向量数据与对应邻接表紧邻存储; (2) 磁盘形式引入了量化压缩机制, 压缩码与码表需驻留于内存中. 因此, 刷新操作主要包括两项任务: (1) 依据磁盘索引的数据布局, 将内存中的向量与邻接表数据持久化至磁盘; (2) 生成量化压缩码和码表文件并写入磁盘.

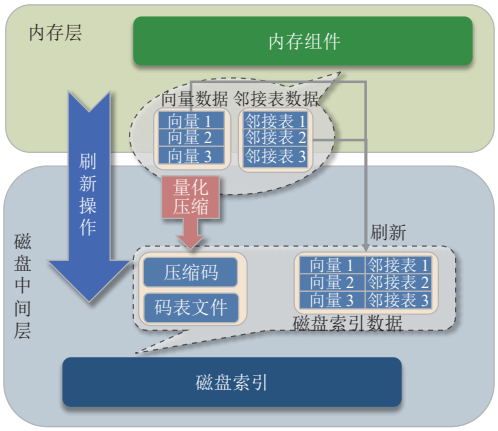


图 6 刷新操作示意图

3.1.5 合并操作

在 3 层索引架构中, 合并操作由原来的“内存层向磁盘层合并”演化为“磁盘中间层向磁盘基本层合并”. 当磁盘中间层组件数量达到设定阈值时, 将触发一次合并操作. 与 FreshDiskANN 相比, 本文所提框架的合并操作在以下 3 方面有所不同: (1) 借助重布局算法 (详见后文第 3.3 节), 删除阶段仅需对磁盘基本层组件进行一趟顺序读操作; (2) 插入阶段的向量来源于磁盘中间层索引, 而非常驻内存, 因此需要从磁盘读取向量数据; (3) 插入点数量显著增多, 为降低内存占用, 不再使用 Δ 结构暂存所有邻接出边.

针对上述挑战, 本文重新设计了合并操作中的删除、插入和修补阶段过程.

(1) 删除阶段: 此阶段负责从磁盘基本层中物理删除被标记删除的点并进行必要修补, 删除流程如算法 2 所示: 首先, 每次从磁盘基本层迭代器获取一个点 (第 2–4 行), 如果该点 ID loc_q 已被标记删除则对该点 x_q 进行实际删除并将其邻居列表 N_q 暂存至 Δ 结构中 (第 5–7 行); 如果该点未被删除, 则遍历其邻居点 n, 若发现被删除邻居, 则将其邻接列表 N_n 合并至当前邻接列表 N_q 中, 并根据 Vamana 剪枝策略淘汰部分节点 (第 8–13 行). 在这个过程

中, 算法会尝试从 Δ 读取邻居点的邻居列表 N_n, 如果其不存在于 Δ 中, 则从磁盘进行读取并加入至 Δ 中 (第 10 行). 如果邻接出边集合有所更新则写回至磁盘.

算法 2. 合并操作删除算法.

输入: 磁盘基本层层级索引 L_1, 增量数据存储结构 Δ.
输出 : *void*

```
1.  begin
2.      Iter_1 ← L_1.GetIterator()                          //初始化磁盘基本层索引迭代器
3.      while Iter_1.HasNext()
4.          (loc_q, x_q, N_q) ← Iter_1.Next()               //依次遍历所有点
5.          if loc_q is deleted
6.              delete x_q and add N_q to Δ                 //实际删除点并缓存邻居列表
7.          else
8.              foreach n in N_q
9.                  if n is deleted
10.                     if Δ not contains N_n then read N_n from disk and add to Δ   //若缓存未命中则读取磁盘
11.                     N_q ← RobustPrune((N_q / {n}) ∪ N_n)                         //对删除点进行修补
12.                 endif
13.             endfor
14.         endif
15.     endwhile
16. end
```

(2) 插入阶段: 如图 7 合并操作插入阶段示意图所示, 本文设计了两个迭代器, 分别是磁盘索引的全量扫描迭代器和多磁盘索引间迭代器, 插入流程如算法 3 所示: 首先, 每次从磁盘中间层迭代器获取一个需要插入的点 (第 4–5 行); 然后, 从磁盘基本层迭代器获取一个空闲的可插入位置 (第 6–12 行), 如果获取到了一个可插入位置 *loc*, 则以插入点 x_q 为查询向量在磁盘基本层索引中获取新的邻接出边集合 (第 13–16 行); 最后, 将插入点向量 x_q 及其邻接出边集合 N_q 插入至磁盘基本层索引的可插入位置 (第 17 行), 并且同时将每条邻接出边的反向边暂存至 Δ 结构中 (第 18 行).

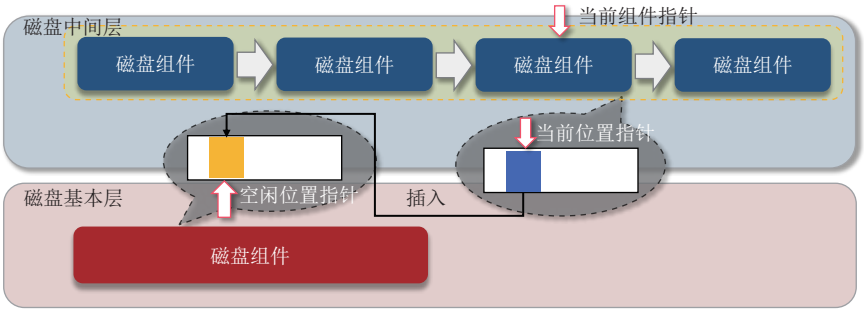


图 7 合并操作插入阶段示意图

算法 3. 合并操作插入算法.

输入: 磁磁盘中间层层级索引 L_0, 磁盘基本层层级索引 L_1, 增量数据存储结构 Δ;
输出: *void*.

```
1.  begin
2.      Iter_0 ← L_0.GetIterator()                  //初始化磁盘中间层索引迭代器
3.      Iter_1 ← L_1.GetIterator()                  //初始化磁盘基本层索引迭代器
4.      while Iter_0.HasNext()
5.          (loc_q, x_q, N_q) ← Iter_0.Next()       //获取一个需要插入的点
6.          loc ← −1                                //初始化插入位置
7.          while Iter_1.HasNext()                  //获取一个空闲位置
8.              (loc_temp, x_temp, N_temp) ← Iter_1.Next()
9.              if loc_temp is free
10.                 loc ← loc_temp
11.             endif
12.         endwhile
13.         if loc == −1
14.             break
15.         endif
16.         N_q ← KNNQuery(L_1, x_q)                //获取插入点的新邻接出边
17.         insert x_q and N_q into L_1's loc position  //进行实际插入
18.         Δ ← Δ ∪ backward edges of N_q           //缓存反向边
19.     endwhile
20. end
```

(3) 修补阶段: 本阶段负责将 Δ 结构中暂存的反向边尝试插入至磁盘基本层索引中, 修补流程如算法 4 所示: 首先, 每次从磁盘中间层迭代器获取一个点 (第 2–4 行), 如果该点 x_q 需要更新则尝试将 Δ 结构中对应的反向边 $\Delta[loc_q]$ 插入至该点的邻接出边集合 N_q 中 (第 5–8 行), 随后, 根据 Vamana 剪枝策略淘汰部分节点 (第 9 行).

算法 4. 合并操作修补算法.

输入: 磁盘基本层层级索引 L_1, 增量据存储结构 Δ;

输出: *void*.

```
1.  begin
2.      Iter_1 ← L_1.GetIterator()          //初始化磁盘基本层索引迭代器
3.      while Iter_1.HasNext()
4.          (loc_q, x_q, N_q) ← Iter_1.Next()   //依次遍历所有点
5           if Δ[loc_q] is empty
6.              continue
7.          endif
8.          N_q ← N_q ∪ Δ[loc_q]
9.          N_q ← RobustPrune(x_q, N_q)     //对反向边进行修补
10.     endwhile
11. end
```

3.1.6 刷新/合并操作 I/O 分析

刷新操作主要包括数据写入与量化压缩两个阶段. 由于磁盘索引布局仅在内存索引数据基础上调整了存储顺序, 数据写入阶段几乎等价于内存索引的直接落盘; 而量化压缩阶段的数据量小, 通常能在秒级完成.

如图 4 所示, 相比于 FreshDiskANN 需要对磁盘基本层索引组件进行 3 趟顺序读和 2 趟顺序写, 虽然本文合

并操作插入阶段需要对所有磁盘中间层索引组件进行 1 趟顺序读操作, 但凭借优化的删除阶段算法与磁盘重布局机制 (详见后文第 3.3 节), 在对磁盘基本层索引组件进行合并操作时只需要 2 趟顺序读操作、少量随机读操作和 2 趟顺序写操作, 却能达到与 FreshDiskANN 合并操作相近的合并 I/O 操作代价. 此外, 上述合并操作的 3 个阶段的算法能很容易地扩展为批处理模式, 即每次处理一批节点, 进一步降低 I/O 开销并提升运行效率.

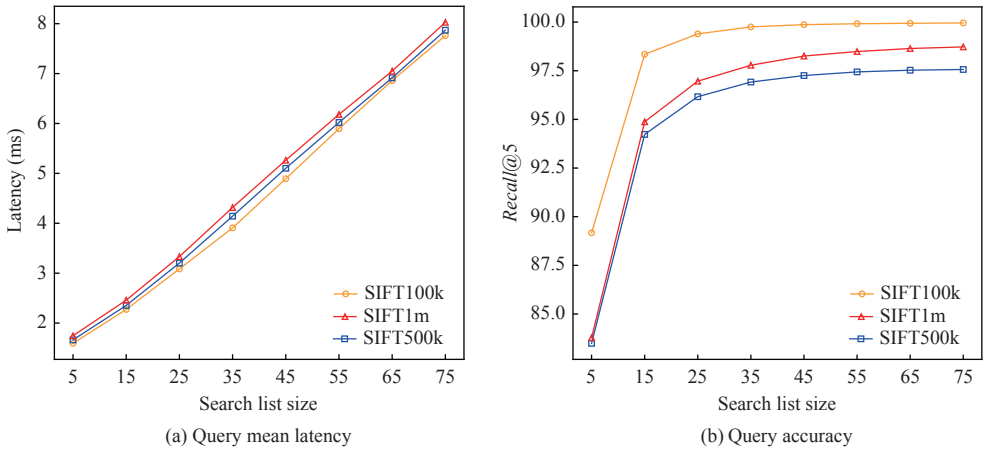
3.1.7 合并策略

本文设计了两种典型的合并触发策略: (1) 写半合并策略: 当插入线程导致磁盘中间层组件数量超过其上限的一半时, 触发向磁盘基本层的合并操作; (2) 定期合并策略: 维护一个后台合并检测线程, 定期检查中间层组件数量, 并在超出阈值后启动合并操作.

3.2 磁盘组件搜索参数动态确定机制设计

(1) 层间动态搜索参数策略

如图 8(a) 所示, 在单个磁盘层组件中, 影响查询延迟的关键因素是搜索列表长度 L_s 而非组件数据量大小, 并且该延迟随 L_s 呈线性增长趋势. 因此, 降低查询延迟最直接有效的方式即为缩小搜索列表长度 L_s. 同时, 如图 8(b) 所示, 查询召回率随搜索列表长度 L_s 的增加呈现对数增长, 且在不同规模数据集上具有一致趋势. 因此, 上述两种指标之间的增长趋势差异, 为实现二者的平衡提供了优化空间.



(a) Query mean latency (b) Query accuracy

图 8 搜索列表长度影响趋势图

基于“越靠近下层, 磁盘组件的数据量越大”这一客观规律, 结合“数据量越大的组件更可能包含最终结果”的启发式思想, 本文设计了一种分层搜索参数策略: 对磁盘中间层和磁盘基本层的组件设定不同的搜索参数 $[L_{s_0}, L_{s_1}]$, 其中满足 $L_{s_0} < L_{s_1}$.

(2) 层内动态搜索参数策略

在 DiskANN 中, 每个磁盘层组件在构建时都要经历如第 2.2 节所言的量化压缩. 由于量化压缩基于聚类方法, 使用聚类中心对整个向量集合进行离散化处理, 因此这些聚类中心本质上反映了组件的局部数据分布特征.

基于这个观察, 受 KRT 导航算法[52]启发, 本文将磁盘中间层组件 I 量化压缩中的聚类中心集合视为其特征点集合 C, 定义向量 x_q 与磁盘中间层组件 I 的组件距离为向量 x_q 到该组件特征点集合 C 中的最小值:

$$Dist(x_q, I) = Dist(x_q, C) = \min_{c \in C} Dist(x_q, c) \tag{2}$$

进一步借鉴 SPANN[11]中的动态决定搜索列表数策略, 在磁盘中间层组件的搜索中, 本文定义了向量 x_q 在给定的 n 个磁盘层组件 I 中的搜索距离阈值公式, 用于判定组件是否值得深入查询, 其中 η 是动态搜索阈值系数:

$$D_{th} = \min_{i \in [0,n)} Dist(x_q, I_i) \times \eta,\ \eta \in (0, 1] \tag{3}$$

由此, 提出磁盘中间层的搜索算法 (算法 5): 首先初始化各层中间结果集并准备好各层级索引查询所需要排除的删除键集合 D (第 2–9 行), 然后再对各层级索引分别进行 K 近邻查询 (第 10–16 行), 其中给定一次 K 近邻查询, 对于组件距离小于搜索阈值 D_{th} 的组件, 则认为更有可能包含最终结果, 则使用本层的标准搜索参数 $L_s = L_{s_0}$; 对于组件距离大于搜索阈值 D_{th} 的组件, 则认为不太可能包含最终结果, 则使用最低限度的搜索参数 $L_s = K$ 以保证搜索的理论正确性 (第 11–14 行). 最后再把过滤后的结果进行聚合并返回 (第 17–19 行).

算法 5. 磁盘中间层搜索算法.

输入: 查询向量x_q, 返回结果集大小K, 磁盘中间层层级索引L_0, 磁盘中间层搜索列表长度参数L_{s_0};

输出: 最终结果集V.

1. **begin**
2. $R \leftarrow [\varnothing$ for i in $0..n-1]$ //初始化中间结果集数组
3. $V \leftarrow \varnothing$ //初始化最终结果集V
4. $I = [I_0, \ldots, I_{n-1}] \leftarrow L_0.GetIndexes()$ //获取磁盘中间层索引组件
5. $I \leftarrow$ sort I by each version number DESC //按照生命周期从新到老排序
6. $D \leftarrow [\varnothing$ **for** i in $0..n-1]$ //初始化删除键集合
7. **for** $i \leftarrow 1$ to $n-1$ //按照生命周期从新到老构建删除键集合
8. $D_i \leftarrow D_{i-1} + I_{i-1}.GetDeleteSet()$
9. **endfor**
10. **for** $i \leftarrow 0$ to $n-1$
11. $L_s \leftarrow L_{s_0}$
12. **if** $Dist(x_q, I_i) > D_{th}$ //动态决定搜索参数
13. $L_s \leftarrow K$
14. **endif**
15. $R_i \leftarrow KNNQuery(I_i, x_q)$ //对各组件分别进行KNN查询
16. **endfor**
17. **for** $i \leftarrow -1$ to 1 // 将查询结果进行聚合
18. $V \leftarrow V \cup R_i$
19. **endfor**
20. **end**

3.3 面向合并操作删除阶段的磁盘索引重布局算法

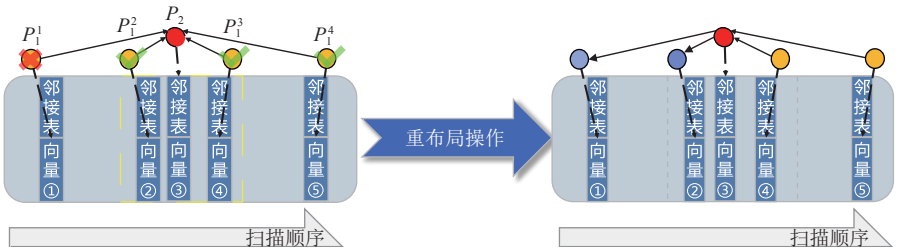
如第 3.1.6 节所述, FreshDiskANN 中的合并操作需要 3 趟顺序读操作, 其中删除阶段需要 2 趟顺序读操作, 从而引发大量磁盘 I/O 操作.

回顾一下第 2.3 节对合并操作删除阶段过程的描述, 假如所有数据均位于内存中, 理论上仅需扫描 1 趟即可完成. 但由于未删除点 P_1 的邻居可能包含被标记为删除的点 P_2, 而 P_1 与 P_2 并不总是同时被加载进内存, 如图 9(a) 所示, 当 P_1 处于 P_1^1 位置, 目前迭代器遍历至黄色区域内的删除点 P_2, 此时修补 P_1 则需要进行一次随机读. 因此原方法必须进行额外的一轮全量读取以避免可能引发的大量随机读.

为了解决该问题, 如图 9(a) 所示, 由于顺序读取带来的先后顺序, 当 P_1 处于位置 P_1^2、P_1^3 和 P_1^4 时, 无须引发一次随机读操作. 基于上述观察, 改进的目标是尽量让 P_2 与 P_1 同时或之前被顺序读入内存, 即如图 9(b) 所示, 希望每个点的物理位置在它的邻接入边邻居前且在它的邻接出边邻居后, 以减少随机读取操作并提升效率.

受 Starling[26]中面向查询操作的磁盘重布局算法启发, 本文提出了一种面向合并操作删除阶段的磁盘索引重布局算法. 与 Starling 中面向的优化场景是索引搜索操作不同, 本文的重布局算法聚焦于合并操作删除阶段的优

化: (1) Starling 在磁盘布局算法中的最小单元是块 (block), 而本文的合并操作中, 每次是从磁盘索引顺序读取一批块, 所以本算法的最小单元是缓冲区 (buffer); (2) Starling 中的优化目标是点 P_1 与邻居 P_2 需要位于同一个最小单元上, 而因为合并操作中顺序读取带来的先后顺序, 所以本算法的优化目标可以放松为 P_2 存在于与 P_1 相同或者之前的最小单元上.



(a) 重布局操作前 (b) 重布局操作后

图 9 重布局操作示意图

为了量化这一目标, 本文给出如下定义.

定义 5 (磁盘索引布局). 给定一个磁盘索引 I, 其向量索引数据在磁盘上的排列顺序称为磁盘索引布局.

定义 6 (松弛化重叠比例). 点 p 的松弛化重叠比例为其入边邻居在当前及后续缓冲区中出现的比例, 假设整个磁盘索引数据在一次全量顺序扫描中分为 0 至 n–1 共 n 段缓冲区进行读取, 给定一个点 p, $BufIdx(p)$ 表示点 p 在第几个缓冲区, $Buf(i)$ 表示第 i 个缓冲区中的点集合, $IN(p)$ 表示点 p 的入边邻居集合, 从而有如下计算公式衡量点 p 的松弛化重叠比例 (relaxed overlap ratio, ROR):

$$ROR(p)=\frac{\sum_{i=BufIdx(p)}^{n-1}|Buf(i)\cap IN(p)|}{|IN(p)|} \tag{4}$$

定义 7 (磁盘索引重布局). 给定一种磁盘索引布局 P, 磁盘索引重布局的目标则是最大化 $ROR(P)$, 其中:

$$ROR(P)=\frac{\sum_{p\in P}ROR(p)}{|P|} \tag{5}$$

基于上述目标, 本文设计了 3 种磁盘索引重布局算法: 第 1 种是最直接的基于入度的重排序算法, 第 2 种是基于贪心的朴素算法, 第 3 种是基于邻居频率启发式的收敛优化算法.

(1) 分配算法 A: 基于入度的排序分配算法

因为每个点的出度是一致且是预先设定的值, 而入度则因为剪枝规则的存在而大相径庭. 将入度大的点排列在前面从直观上更容易提高其入边邻居出现在之后缓冲区的概率, 从而提升松弛化重叠比例. 因此, 不妨考虑按照每个点的邻接入边数 (即入度) 进行降序排序, 并按照该顺序进行索引重布局.

(2) 分配算法 B: 基于贪心的朴素分配算法

在分配算法 A 的布局基础上, 本文进一步提出基于贪心的朴素分配算法, 期望在尽量不破坏分配算法 A 的先后顺序基础上, 进一步加强松弛化条件. 具体流程如算法 6 所示: 按照分配算法 A 中的排序顺序访问所有点, 当访问点 p_2 时, 该算法将 p_2 尽可能分配到原本的分区 (第 5–8 行), 并将其入边邻居 p_1 贪心地分配到原本的分区和 p_2 的分区中的下标较大值分区 (第 9–14 行).

算法 6. 基于贪心的朴素分配算法.

输入: 图$G=(V,IN(V))$;

输出: 新磁盘索引布局P.

```
begin
M ← mapping of vertex IDs to previous partition IDs
P = [P_0, ..., P_{n-1}] ← [∅ for i in 0..n − 1]          //清空原本的布局
foreach v in V
    x ← M[v]                                              //尝试分配至原布局位置
    while x < n and P_x is full then x ← x + 1
    if x = n then x ← any free partition
    P_x ← P_x ∪ v
    foreach u ∈ IN(v)
        y ← max(M[u], x)                                  //尝试分配邻居至原布局位置
        while y < n and P_y is full then y ← y + 1
        if y = n then y ← any free partition
        P_y ← P_y ∪ u
    endfor
endfor
end
```

(3) 分配算法 C: 基于邻居频率启发式的收敛分配算法

在分配算法 B 的布局基础上, 本文再提出基于邻居频率启发式的收敛优化算法, 期望在尽量不破坏分配算法 B 的先后顺序基础上, 进一步加强松弛化条件, 具体流程如算法 7 所示: 给定迭代轮数β与已有布局 (第 2–4 行), 按照随机顺序访问所有点 (第 5 行), 当访问点 p 时, 统计其邻接出边点所在分区号的最大值$\mathrm{PID}_{\mathrm{direct}}$与邻接入边点所在分区号的最小值$\mathrm{PID}_{\mathrm{reverse}}$ (第 6–7 行), 如果$\mathrm{PID}_{\mathrm{direct}}$小于等于$\mathrm{PID}_{\mathrm{reverse}}$, 那么当前访问点 p 则可以取到局部最优解, 也就是把点 p 分配到$\mathrm{PID}_{\mathrm{direct}}$与$\mathrm{PID}_{\mathrm{reverse}}$之间的最小分区上 (第 9–10 行); 否则, 则把点分配到$\mathrm{PID}_{\mathrm{reverse}}$与$\mathrm{PID}_{\mathrm{direct}}$之间权重最大值分区上 (第 11–21 行), 权重值为当前分区及以后的入边邻居数与当前分区及以前的出边邻居数之和 (第 13–16 行).

算法 7. 基于启发式的收敛分配算法.

输入: 图$G = (V, N(V))$, 迭代轮数β;

输出: 新布局P.

```
1.  begin
2.  M ← mapping of vertex IDs to previous partition IDs
3.  P = [P_0, ..., P_{n-1}] ← [∅ for i in 0..n − 1]          //清空原本的布局
4.  while iteration ⩽ β
5.      foreach v in V
6.          PID_direct ← max partition ID in direct neighbors
7.          PID_reverse ← min partition ID inreverse neighbors
8.          x ← −1
9.          if PID_direct ⩽ PID_reverse
10.             x ← ID of smallest partition ∈ [P_{PID_direct}, P_{PID_reverse}]
11.         else
12.             W_max ← −1
13.             for i ← PID_reverse to PID_direct
```

```
            W_direct ← direct neighbors number before P_i
            W_reverse ← reverse neighbors number after P_i
            W_i ← W_direct + W_reverse
            if P_i is not full and W_i > W_max
              W_max ← W_i; x ← i
            endif
          endfor
        endif
        if x = −1 then x ← any free partition          //分配一个空闲分区
        P_x ← P_x ∪ v
        endfor
      endwhile
end
```

3.3.1 I/O 分析与重布局策略

上述 3 种分配算法的效率虽然较高, 但在应用至系统中时, 仍会引发较大的短期 I/O 开销: (1) 获取原始磁盘布局信息以及邻接图信息需要对磁盘基本层索引组件进行全量顺序读操作; (2) 在获取到磁盘重布局位置映射信息后, 根据布局映射重写磁盘索引时需进行大量随机读与全量顺序写.

虽然重布局会产生大量 I/O 操作, 但是仍旧有以下理由支持执行重布局操作: 无论是磁盘合并操作, 还是磁盘重布局操作, 都是一种低频、高代价但能获取显著长期收益的操作, 其目的都是为了降低操作的冗余性, 合并操作是为了降低磁盘中间层在查询操作时的冗余性而重布局操作则是为了降低磁盘基本层在合并操作时的冗余性. (1) 在实际生产环境中, 只要磁盘中间层存在数据, 合并操作则可以在查询操作空闲时进行; 同理, 只要磁盘基本层存在数据, 重布局操作则可以在合并操作与查询操作空闲时进行, 进而可以降低重布局操作本身引发的大量 I/O 操作对系统性能的影响. (2) 只有磁盘中间层保存的数据量足够大时, 才会触发一次合并操作; 同理, 重布局操作仅在磁盘基本层的松弛化重叠比例低于阈值时才被触发, 从而能有效控制其频率, 较低的重布局操作频率淡化了其本身引发大量 I/O 对系统性能的影响.

本文引入了如下策略用于判断是否触发重布局.

(1) 测试物理机当前使用磁盘的随机读速度 V_r 与顺序读速度 V_s, 并设置一个重布局阈值系数 λ, 如公式 (6) 所示:

$$\lambda = \frac{V_r}{V_s},\ \lambda \in (0, 1] \tag{6}$$

(2) 监控每轮合并操作中删除阶段产生的随机读块数 B_r 与顺序读块数 B_s 的比率, 用于近似表示松弛化重叠比例, 当满足公式 (7) 时, 则触发一次重布局操作:

$$\frac{B_r}{B_s} \geqslant \lambda \tag{7}$$

4 实验分析

4.1 实验数据

我们在多个公开向量数据集上进行了实验, 包括 SIFT1M、GIST1M 和 DEEP10M. 表 1 展示了每个数据集的详细信息.

● SIFT1M[53]是一个经典的图像向量数据集, 用于评估支持大规模近似最近邻搜索 (approximate nearest neighbor search, ANNS) 算法的性能. 该数据集由图像的 SIFT (尺度不变特征变换) 特征构建, 包含 100 万个 128 维向量和 1 万个查询向量.

表 1 实验数据集

数据集	向量维度	向量距离类型	向量类型	向量数量	查询数量
SIFT1M	128	L2	float	1 000 000	10 000
GIST1M	960	L2	float	1 000 000	1 000
DEEP10M	96	L2	float	10 000 000	10 000

● GIST1M[53]同样为评估大规模 ANNS 性能的图像数据集, 维度更高, 由图像的全局特征描述符生成, 包含 100 万个 960 维向量及 1 000 个查询向量.

● DEEP10M[39]是一个规模更大的图像向量数据集. 图像经过 GoogLeNet 模型处理并通过 PCA 降维至 96 维, 最终包含 1 000 万个 96 维向量及 1 万个查询向量.

4.2 评价指标及基准方法

根据 FreshKANN 的定义, 我们采用以下 4 项指标来评估性能: (1) 查询召回率 (*Recall*), 用于衡量查询精度; (2) 查询操作 QPS, 每秒完成的查询操作数, 用于反映查询吞吐量; (3) 插入操作 QPS, 每秒完成插入操作数, 用于反映更新吞吐量; (4) 内存用量变化, 用于评估硬件资源消耗.

本文将所提方法 LSMDiskANN 与当前领先的可更新图结构磁盘向量索引算法 FreshDiskANN (FreshDiskANN 代码仓库: https://github.com/microsoft/DiskANN/tree/diskv2) 进行对比. 两者在公共参数设置上保持一致, 确保实验公平性. 尽管近期已有针对 FreshDiskANN 的改进工作发表于 arXiv[25,46], 但由于未开源, 故未纳入对比对象.

4.3 实验设置

本文所有的实验都在同一台配置为 Intel(R) Xeon(R) Platinum 8352V @ 2.10 GHz 的 CPU, 128 GB 的 DDR4 内存和 1 TB 的 SSD 硬盘 (顺序读速度 $V_r = 567$ MB/s, 随机读速度 $V_s = 390$ MB/s) 的服务器上完成.

本文设计了两个对比实验和两个消融分析, 其中对比实验模拟实际应用场景中向量索引的使用情况, 设计了索引快速膨胀场景和索引稳定更新场景 (代码已发布在 https://github.com/N0ir7/LSMDiskANN).

(1) 实验 1: 索引快速膨胀实验. 首先, 从原始数据集中抽取 10% 数据构建初始索引, 随后经历 100 个迭代轮次, 数据量膨胀到原始数据集 80% 的数据. 在每个迭代轮次中, 会不断进行插入、查询与必要的合并操作, 并保证在每个迭代轮次中随机插入原数据集总量 0.7% 的数据.

(2) 实验 2: 索引稳定更新实验. 在实验 1 的索引基础上, 整个索引再经历 100 个迭代轮次. 在每个迭代轮次中, 会不断进行插入、删除查询与必要的合并操作, 并保证在每个迭代轮次中各随机插入与删除原数据集总量 0.3% 的数据.

消融实验则分别在实验 1 和实验 2 的场景下, 对本文提出的优化策略进行有效性验证.

在本文实验中, 对于相同数据集的不同实验均采用相同的参数. 对于不同方法之间的共有参数, 也采用相同参数, 并使用 FreshDiskANN 中的默认设置. 表 2 共同实验参数展示了不同方法之间的共用参数, 表 3 展示了 LSMDiskANN 的特有参数.

表 2 共同实验参数

查询线程数	插入线程数	删除线程数	邻居列表长度 R	搜索参数 L_s	剪枝系数 α	合并时最小单元大小 (MB)
6	2	1	63	75	1.2	256

表 3 LSMDiskANN 实验参数

内存层组件数	磁盘中间层组件数最大值	磁盘中间层合并组件数阈值	刷新线程检查频率	合并线程检查频率	磁盘中间层搜索参数 L_{s_0}	动态搜索阈值系数 η	重布局阈值系数 λ
2	5	3	5 s	20 s	15	1.6	0.68

说明: 磁盘中间层搜索参数 L_{s_0} 取值依据图 8(b) 曲线斜率变化, 在搜索列表长度为 15 时, 曲线斜率达最大值, 且召回率满足基本要求; 动态搜索阈值系数参考 SPANN[11]进行设置; 重布局阈值系数 λ 由公式 (6) 确定.

4.4 实验结果与分析

4.4.1 对比实验

本文两个对比实验均在每个数据集上进行了 100 轮次的实验, 并对每个轮次中的多次查询取均值, 从而得到如图 10–图 15 所示的有关查询性能指标实验图, 同时对 100 轮次进行取平均, 从而得到如表 4、表 5 的汇总表. 由于删除操作较为简单, 每个批次的删除操作在不同方法中均在毫秒级完成, 不具有比较意义, 因此, 本文实验分析将删除操作的吞吐量分析省去.

为了评估 LSMDiskANN 的有效性, 本文围绕以下 5 个关键问题展开分析.

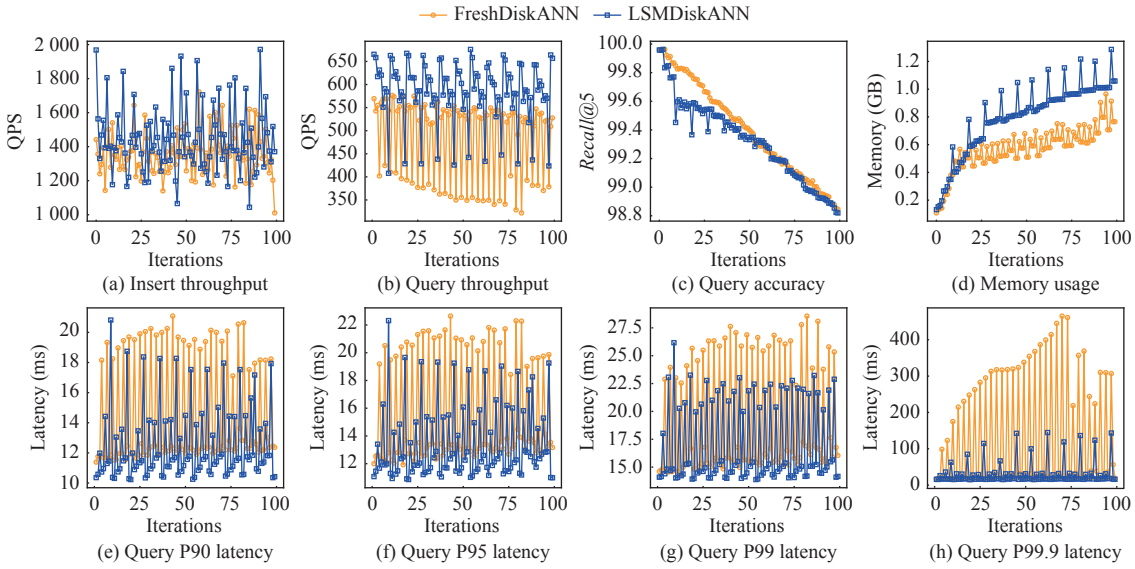


图 10 SIFT1M 数据集下索引快速膨胀实验总体表现

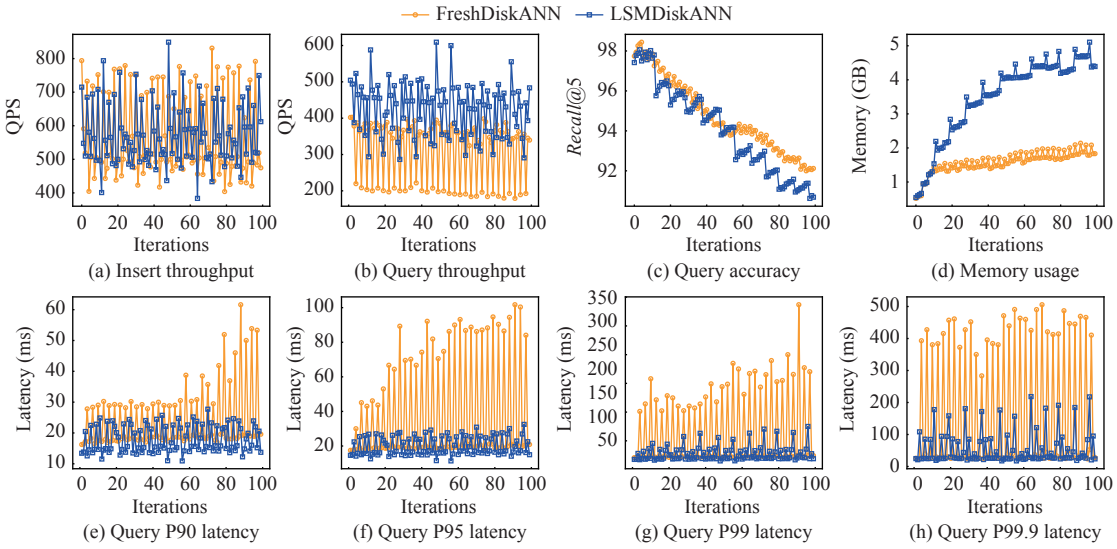


图 11 GIST1M 数据集下索引快速膨胀实验总体表现

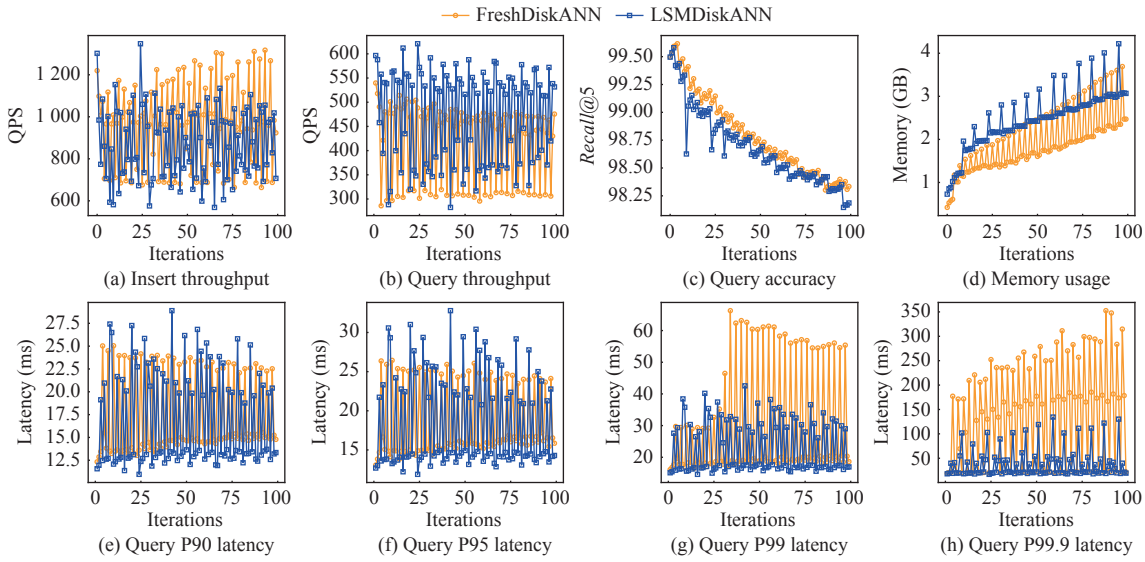


图 12 DEEP10M 数据集下索引快速膨胀实验总体表现

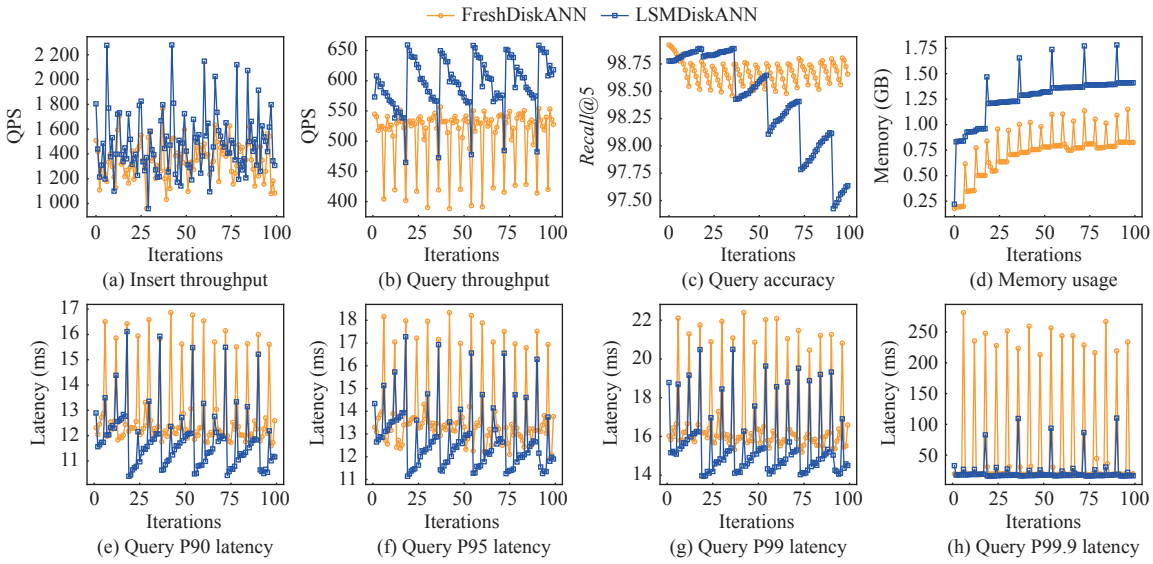


图 13 SIFT1M 数据集下索引稳定更新实验总体表现

● Q1: LSMDiskANN 是否可以获得比 FreshDiskANN 更高的查询吞吐量?

从图 10–图 15 中查询吞吐量变化图来看, 答案是肯定的. 在索引快速膨胀实验中, LSMDiskANN 在 3 项数据集上分别平均取得了 20.76%、35.5% 和 13.24% 的查询 QPS 显著提升; 在索引稳定更新实验中, LSMDiskANN 在 3 项数据集上分别平均取得了 16.6%、19.22% 和 11.73% 的查询 QPS 显著提升.

该提升归因于合并操作频率的下降, 正如前文所言, FreshDiskANN 频繁的合并操作占用了大量系统资源, 进而严重影响前台查询操作性能. 此外, 因为周期性合并操作的存在, 使得查询吞吐变化呈现波峰-波谷的震荡式变化; LSMDiskANN 将原本的合并操作拆分为刷新与合并操作, 使得变化曲线变为波峰-波腰-波谷的平缓式变化, 系统运行更加平滑, 查询吞吐量显著上升.

● Q2: LSMDiskANN 是否可以获得比 FreshDiskANN 更高的更新吞吐量?

图 10–图 15 中插入吞吐量变化图结果显示, 相比于 FreshDiskANN, LSMDiskANN 在 SIFT1M 上平均取得了 6.42% 与 14.24% 的插入 QPS 提升, 在 GIST1M 上基本持平, 在 DEEP10M 上略有降低.

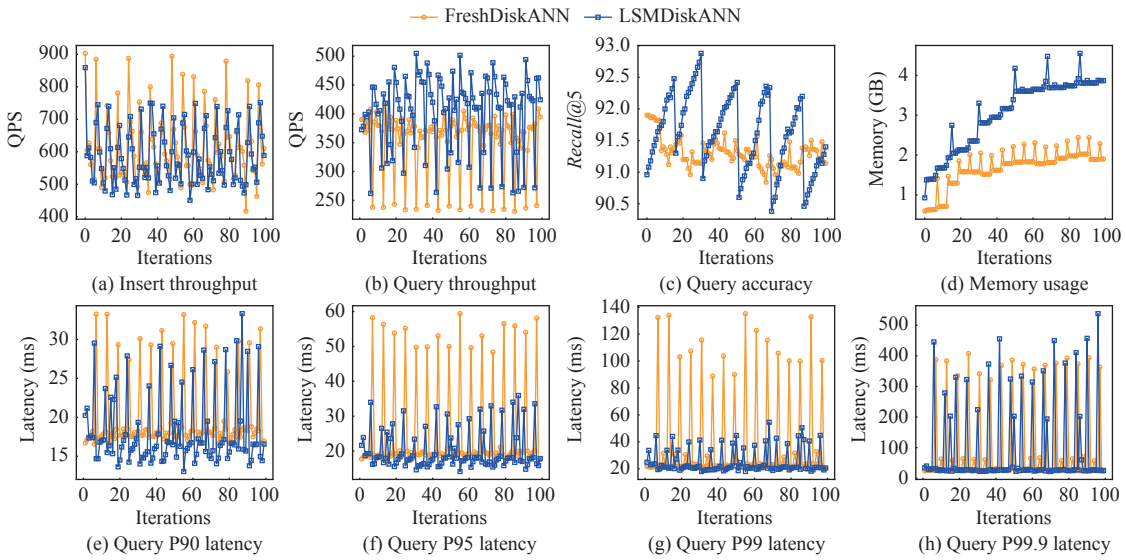


图 14 GIST1M 数据集下索引稳定更新实验总体表现

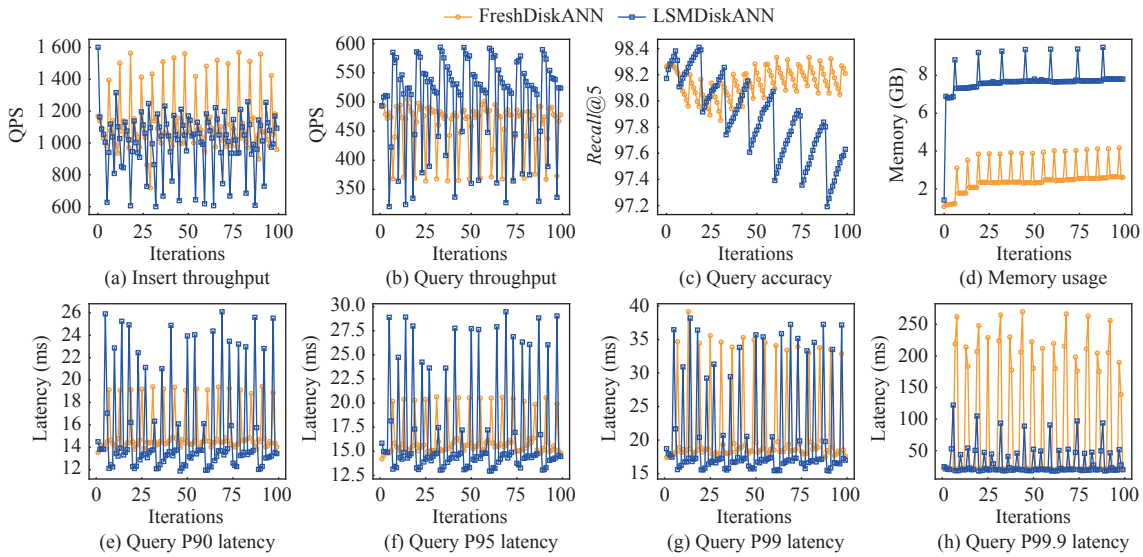


图 15 DEEP10M 数据集下索引稳定更新实验总体表现

表 4 索引快速膨胀实验汇总表

数据集	Insert QPS		QPS		P90 latency		P95 latency		P99 latency		P99.9 latency	
	FDA	LDA	FDA	LDA	FDA	LDA	FDA	LDA	FDA	LDA	FDA	LDA
SIFT1M	1 357.18	**1 444.35**	485.45	**586.25**	14.49	**12.48**	15.50	**13.42**	18.87	**16.99**	114.12	**30.30**
GIST1M	568.35	**574.65**	311.99	**422.77**	23.67	**17.87**	37.71	**19.91**	62.96	**27.70**	158.09	**56.13**
DEEP10M	**964.94**	873.12	417.75	**473.07**	17.28	**16.93**	**18.53**	18.63	28.85	**23.13**	138.49	**39.59**

注: FDA代指FreshDiskANN, LDA代指LSMDiskANN

表 5 索引稳定更新实验汇总表

数据集	Insert QPS		QPS		P90 latency		P95 latency		P99 latency		P99.9 latency	
	FDA	LDA	FDA	LDA	FDA	LDA	FDA	LDA	FDA	LDA	FDA	LDA
SIFT1M	1 349.56	**1 541.78**	511.78	**596.75**	12.83	**11.80**	13.86	**12.82**	16.84	**15.67**	55.82	**21.76**
GIST1M	**608.84**	589.54	353.49	**421.41**	19.86	**17.39**	24.65	**19.22**	37.29	**24.66**	**85.96**	91.20
DEEP10M	**1 116.94**	985.96	459.42	**510.65**	15.13	**14.81**	16.19	**16.15**	21.00	**19.45**	80.07	**28.49**

注: FDA代指FreshDiskANN, LDA代指LSMDiskANN

因为插入操作仅在内存中完成, 受 I/O 操作影响较小, 所以在插入吞吐量变化上, 合并操作的延缓并未显著影响插入 QPS. 同时, 由于内存索引组件数据量小, 其图结构收敛性不稳定, 并随数据量减小而收敛速度加快, 进而直接影响插入操作延迟. 因此插入 QPS 受内存索引转换频率影响较大, 且该转换频率与合并操作频率形成一个取舍关系.

● Q3: LSMDiskANN 是否可以获得比 FreshDiskANN 更稳定的查询表现?

图 10–图 15 的查询延迟分位点变化图结果显示, 在多项查询延迟指标 (P90、P95、P99、P99.9) 上, LSMDiskANN 均显著优于 FreshDiskANN. 例如, 在 SIFT1M 上, 最大可降低查询延迟 73.45%; 在 GIST1M 上, 最大可降低查询延迟 64.5%; 在 DEEP10M 上最大可降低查询延迟 71.42%.

虽然磁盘层多层的设计使得冗余查询增多, 在仅考虑查询操作时, 会不可避免使得查询延迟上升, 但在查询-更新混合负载下, 这种设计使得系统资源的整体状态更加稳定, 变化幅度小, 可用计算资源增多, 进而显著降低了复杂场景下查询的极端查询延迟, 在实际生产环境中, 提供给用户更佳的查询体验.

● Q4: LSMDiskANN 是否可以获得与 FreshDiskANN 相当的召回率?

在两个实验中, LSMDiskANN 与 FreshDiskANN 的召回率总体相当且均处于高位, 差异处于可接受范围内. 在索引快速膨胀实验中 (如图 10–图 12 所示), 考虑到插入数据的随机性, 二者的召回率均在合理的范围内波动; 在索引稳定更新实验中 (如图 13–图 15 所示), 虽然 LSMDiskANN 在实验后期的查询召回率略低于 FreshDiskANN, 但是都处在一个高位召回率水平; 图 14、图 15 显示, LSMDiskANN 与 FreshDiskANN 的平均召回率基本一致, 但是 LSMDiskANN 的波动范围略大一些.

每当新鲜数据插入到上层索引时, 由于冗余查询的存在, 此时召回率会不断上升; 每当数据从上层索引往下层索引进行传递时, 都不可避免的导致召回率的陡然下降, 从而出现召回率波动的现象. 同时, 由于 FreshDiskANN 本身合并操作算法的两点缺陷, 导致 LSMDiskANN 召回率的下降与波动现象更加明显.

(1) 在合并操作后, 不会调整对应量化压缩中聚类中心的分布, 从而弱化了量化压缩反映真实向量数据分布的特性, 并注定了磁盘向量索引在不断合并过程中召回率的下降趋势. 这点在 GIST 数据集上尤为明显, 这是因为 GIST 数据集中向量维度更高, 其量化压缩反映真实向量数据分布的抗干扰性更弱, 进而加剧其召回率劣化速度. 因此, 无论如何调整插入算法, 当插入量积累足够多后, 必须通过重建或者调整量化表来维持高位召回率.

(2) 在合并操作插入阶段时, 所有插入点之间没有连边机制, 因此同一批插入点之间是不能互连的, 从而导致图连通性下降, 但是在下一批次合并操作时, 新批次插入点会成为上一轮批次插入点之间的桥, 从而可能使得图连通性有所弥补, 并出现图 14 中合并操作后的召回率有所回升的现象. 因此, 当插入点批次量越大时, 图质量下降幅度越大. 由于 LSMDiskANN 每次合并操作数据量的增大, 进一步加剧了图劣化速度, 从而导致如图 14、图 15 中更大的波动范围.

总的来说, 相比于 FreshDiskANN, LSMDiskANN 的召回率波动均在可容忍的范围内, 关于合并操作插入算法的改进, 本文将其留作一个未来工作.

● Q5: LSMDiskANN 是否可以获得比 FreshDiskANN 更低的内存占用?

从图 10–图 15 的内存占用变化来看, 结论是否定的. 不可否认的事实是, LSMDiskANN 由于引入了磁盘中间层, 需要维护更多的磁盘层索引组件, 进而带来更多的量化压缩对应的码表, 内存开销必然有所增加. 但更多的内存开销源于磁盘索引组件之间的数据冗余.

由于 FreshDiskANN 原本并未考虑在单个系统中集成多个磁盘索引组件, 因此在现有磁盘索引实现中, 均保留了一份索引执行所需的完整上下文结构, 比如重复的元数据和查询所需的内存上下文. 进而导致在多磁盘索引组件的情况下, 内存占用随着磁盘组件数的上升呈现线性增长趋势. 若能实现组件间共享资源机制, 比如统一元数据管理和线程池管理, 则可大幅降低内存开销. 这类共享资源的方案属于工程实现的范畴, 因此本文并未在此方面做进一步测试.

4.4.2 磁盘组件搜索参数动态确定机制消融实验

为了评估磁盘组件搜索参数动态确定机制的有效性, 本文在 SIFT1M 索引快速膨胀实验设置下, 对层间动态搜索参数策略 (以下简称策略 1) 与层内动态搜索参数策略 (以下简称策略 2) 进行了消融实验分析, 实验选取其中策略 2 生效的查询记录并按迭代轮次取均值从而得到实验结果, 如图 16 所示.

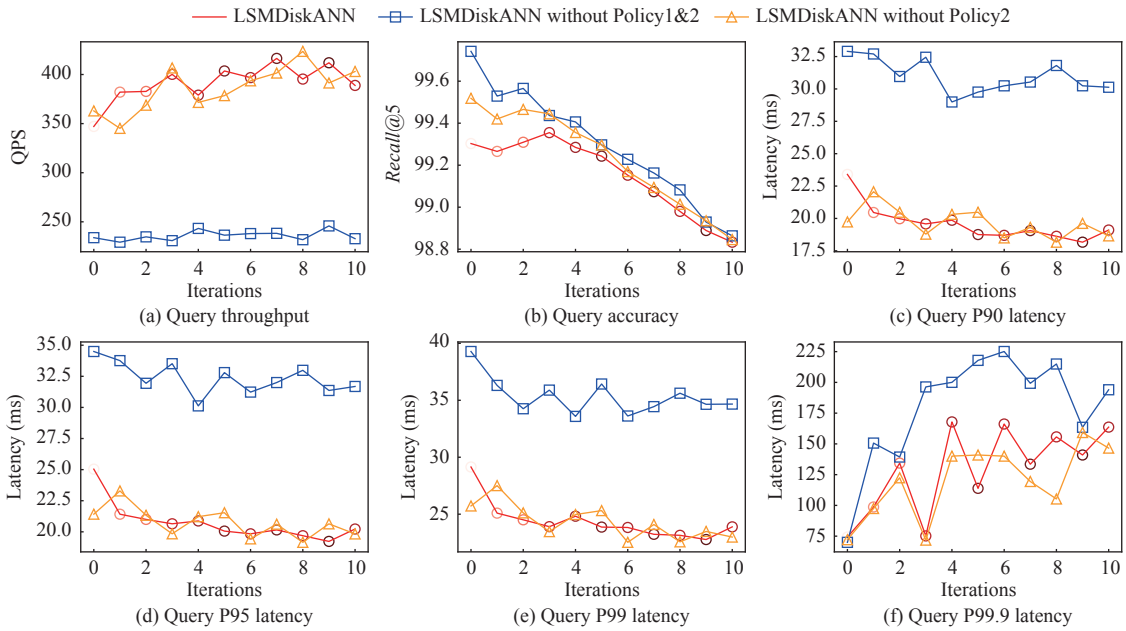


图 16 磁盘组件搜索参数动态确定机制消融实验

从实验结果可以看出, 相较于完全不使用动态搜索参数, LSMDiskANN 在查询 QPS 上平均提升了 65.86%, 在查询延迟 P90、P95、P99、P99.9 上分别平均下降了 36.7%、35.89%、31.01% 和 27.74%. 进一步比较策略 1 与联合使用策略 1 和策略 2 的效果可发现, 后者在查询 QPS 上平均提升了 1.35%, 而在极端查询延迟上几乎持平. 这是由于策略 1 中查询列表长度的默认值为 15, 而策略 2 将其下调至 5, 下降幅度相对较小, 因此带来的性能优化也较有限.

综上所述, 磁盘组件搜索参数动态确定机制在提升查询吞吐量及降低高分位延迟方面均表现出良好的实用性与有效性.

4.4.3 磁盘索引重布局算法消融实验

为了评估磁盘索引重布局算法的有效性, 本文在 SIFT1M 索引稳定更新实验设置下, 比较了基于入度的排序分配 (in degree rank relayout, IDRR) 算法、基于贪心的朴素分配 (greedy allocation relayout, GAR) 算法和基于邻居频率启发式的收敛分配 (neighbor frequency relayout, NFR) 算法这 3 种磁盘重布局算法对松弛化重叠比例 (*ROR*) 的提升效果, 其中 RAW 表示未经任何重布局优化的初始索引布局, 实验效果如图 17 所示.

图 17 展示了具有先后顺序, 编号为 0–2 的最小单元在经过 3 种算法处理后的 *ROR* 提升效果. 由于所有数据点都在最小单元 0 及其之后, 因此最小单元 0 的 *ROR* 恒为 100%. 可以看出: (1) 在最小单元 1 中, 各分位点层级的 *ROR*

显著提升, 平均提升幅度为 42.81%; (2) 在最小单元 2 中, 平均 *ROR* 提升更为明显, 达 163.15%; (3) 从全局来看, 索引总布局的平均 *ROR* 从 67.6% 提升至 83.2%. 这一结果表明, 即使在合并操作的删除阶段需要大量删除磁盘索引点, 在优化后的布局下, 最坏情况下也仅会触发不超过 16.8% 的随机读操作, 从而有效降低了 I/O 操作成本.

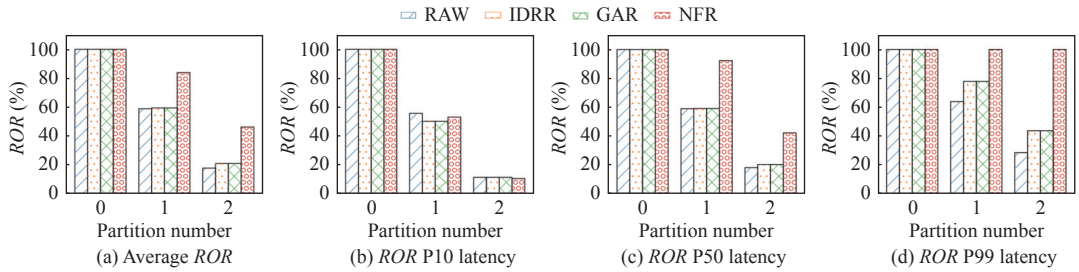


图 17 不同最小单元在不同算法下 *ROR* 提升效果

此外, 本文还对是否启用磁盘索引重布局算法进行了消融实验分析, 并统计了每轮合并操作中删除阶段的磁盘 I/O 操作时间, 实验结果如图 18 所示.

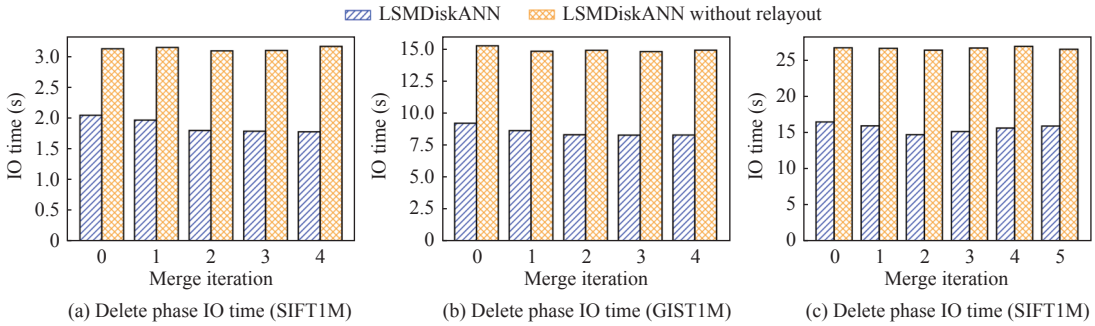


图 18 磁盘索引重布局算法消融实验

如图 18 所示, 在采用重布局算法并结合本文提出的删除流程后, 在 3 个数据集上, 整个删除阶段的磁盘 I/O 时间分别平均下降了 40.06%、42.76%、41.26%, 进一步验证了所提出磁盘索引重布局算法在降低系统开销方面的有效性.

5 总 结

基于 LSM 的次级索引广泛应用于 NoSQL 数据库, 在高吞吐量场景下有效地解决了索引表的同步更新与高并发查询问题, 展现出极强的实用价值. 受此启发, 本文提出了一种基于 LSM 思想的更新友好型磁盘向量索引框架. 针对现有领先算法 FreshDiskANN 在查询-更新混合场景中存在的查询吞吐量瓶颈与极端查询延迟过高等问题, 本文借鉴 LSM 的分层思想并结合 FreshDiskANN 的系统架构, 设计并引入了磁盘中间层, 以缓解访问瓶颈. 同时, 本文进一步提出了磁盘组件搜索参数动态确定机制与面向合并操作删除阶段的磁盘索引重布局算法, 在降低查询延迟与减少合并操作删除阶段 I/O 开销方面取得显著效果. 通过在多个经典大规模高维向量数据集上进行的索引快速膨胀场景与索引稳定更新场景下的对比实验与消融分析, 本文所提出的方法在查询吞吐量以及极端查询延迟方面均有显著改善, 验证了该框架的有效性与实用性.

References

[1] Asai A, Min S, Zhong ZX, Chen DQ. Retrieval-based language models and applications. In: Proc. of the 61st Annual Meeting of the Association for Computational Linguistics, Vol. 6 (Tutorial Abstracts). Toronto: ACL, 2023. 41–46. [doi: 10.18653/v1/2023.acl-tutorials.

6]

[2] Li S, Lv FY, Jin TW, Lin GL, Yang KP, Zeng XY, Wu XM, Ma QL. Embedding-based product retrieval in taobao search. In: Proc. of the 27th ACM SIGKDD Conf. on Knowledge Discovery & Data Mining. Singapore: ACM, 2021. 3181–3189. [doi: 10.1145/3447548.3467101]

[3] Zhu J. Application of vector search in batch retrieval of e-commerce products. Bao-steel Technology, 2023(4): 13–16 (in Chinese with English abstract). [doi: 10.3969/j.issn.1008-0716.2023.04.004]

[4] Indyk P, Motwani R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In: Proc. of the 13th Annual ACM Symp. on Theory of Computing. Dallas: ACM, 1998. 604–613. [doi: 10.1145/276698.276876]

[5] Malkov YA, Yashunin DA. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Trans. on Pattern Analysis and Machine Intelligence, 2020, 42(4): 824–836. [doi: 10.1109/TPAMI.2018.2889473]

[6] Ono N, Matsui Y. Relative NN-descent: A fast index construction for graph-based approximate nearest neighbor search. In: Proc. of the 31st ACM Int'l Conf. on Multimedia. Ottawa: ACM, 2023. 1659–1667. [doi: 10.1145/1963405.1963487]

[7] Dong W, Moses C, Li K. Efficient k-nearest neighbor graph construction for generic similarity measures. In: Proc. of the 20th Int'l Conf. on World Wide Web. Hyderabad: ACM, 2011. 577–586. [doi: 10.1145/1963405.1963487]

[8] Harwood B, Drummond T. FANNG: Fast approximate nearest neighbour graphs. In: Proc. of the 2016 IEEE Conf. on Computer Vision and Pattern Recognition. Las Vegas: IEEE, 2016. 5713–5722. [doi: 10.1109/CVPR.2016.616]

[9] Jégou H, Douze M, Schmid C. Product quantization for nearest neighbor search. IEEE Trans. on Pattern Analysis and Machine Intelligence, 2011, 33(1): 117–128. [doi: 10.1109/TPAMI.2010.57]

[10] Datar M, Immorlica N, Indyk P, Mirrokni VS. Locality-sensitive hashing scheme based on p-stable distributions. In: Proc. of the 20th Annual Symp. on Computational Geometry. Brooklyn: ACM, 2004. 253–262. [doi: 10.1145/997817.997857]

[11] Chen Q, Zhao B, Wang HD, Li MQ, Liu CJ, Li ZZ, Yang M, Wang JD. SPANN: Highly-efficient billion-scale approximate nearest neighbor search. In: Proc. of the 35th Int'l Conf. on Neural Information Processing Systems. Curran Associates Inc., 2021. 398.

[12] Dasgupta S, Freund Y. Random projection trees and low dimensional manifolds. In: Proc. of the 40th Annual ACM Symp. on Theory of Computing. Victoria: ACM, 2008. 537–546. [doi: 10.1145/1374376.1374452]

[13] Jayaram Subramanya S, Devvrit, Kadekodi R, Krishaswamy R, Simhadri HV. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2019. 1233.

[14] Ren J, Zhang MJ, Li D. HM-ANN: Efficient billion-point nearest neighbor search on heterogeneous memory. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 895.

[15] Abu-El-Haija S, Kothari N, Lee J, Natsev P, Toderici G, Varadarajan B, Vijayanarasimhan S. YouTube-8M: A large-scale video classification benchmark. arXiv:1609.08675, 2016.

[16] Priem J, Piwowar H, Orr R. OpenAlex: A fully-open index of scholarly works, authors, venues, institutions, and concepts. arXiv:2205.01833, 2022.

[17] Xu YM, Liang HY, Li J, Xu ST, Chen Q, Zhang QX, Li C, Yang ZY, Yang F, Yang YQ, Cheng P, Yang M. SPFresh: Incremental in-place update for billion-scale vector search. In: Proc. of the 29th Symp. on Operating Systems Principles. Koblenz: ACM, 2023. 545–561. [doi: 10.1145/3600006.3613166]

[18] Singh A, Subramanya SJ, Krishnaswamy R, Simhadri HV. FreshDiskANN: A fast and accurate graph-based ANN index for streaming similarity search. arXiv:2105.09613, 2021.

[19] Mohoney J, Pacaci A, Chowdhury SR, Minhas UF, Pound J, Renggli C, Reyhani N, Ilyas IF, Rekatsinas T, Venkataraman S. Incremental IVF index maintenance for streaming vector search. arXiv:2411.00970, 2024.

[20] Wei CX, Wu B, Wang S, Lou RJ, Zhan CQ, Li FF, Cai YZ. AnalyticDB-V: A hybrid analytical engine towards query fusion for structured and unstructured data. Proc. of the VLDB Endowment, 2020, 13(12): 3152–3165. [doi: 10.14778/3415478.3415541]

[21] Wang JG, Yi XM, Guo RT, Jin H, Xu P, Li SJ, Wang XY, Guo XZ, Li CM, Xu XH, Yu K, Yuan YX, Zou YH, Long JQ, Cai YD, Li ZX, Zhang ZF, Mo YH, Gu J, Jiang RY, Wei Y, Xie C. Milvus: A purpose-built vector data management system. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 2614–2627. [doi: 10.1145/3448016.3457550]

[22] Qdrant. High-performance vector search at scale. 2025. https://qdrant.tech/

[23] Weaviate. For AI engineers who think big. 2025. https://weaviate.io/

[24] Vardhan Simhadri H, Aumüller M, Ingber A, Douze M, Williams G, Dobson Manohar M, Baranchuk D, Liberty E, Liu F, Landrum B, Karjikar M, Dhulipala L, Chen M, Chen Y, Ma R, Zhang K, Cai YZ, Shi JY, Chen YZ, Zheng WG, Wan ZH, Yin J, Huang B. Results of the big ANN: NeurIPS'23 competition. arXiv:2409.17424, 2024.

[25] Yu S, Lin SY, Gong SF, Xie YQ, Liu RC, Zhou YJ, Sun J, Zhang YF, Li GL, Yu G. A topology-aware localized update strategy for graph-based ANN index. arXiv:2503.00402, 2025.

[26] Wang MZ, Xu WZ, Yi XM, Wu SL, Peng ZY, Ke XY, Gao YJ, Xu XL, Guo RT, Xie C. Starling: An I/O-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. Proc. of the ACM on Management of Data, 2024, 2(1): 14. [doi: 10.1145/3639269]

[27] Pan JJ, Wang JG, Li GL. Survey of vector database management systems. The VLDB Journal, 2024, 33(1): 1591–1615.

[28] Aumüller M, Bernhardsson E, Faithfull A. ANN-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. Information Systems, 2020, 87: 101374. [doi: 10.1016/j.is.2019.02.006]

[29] Li W, Zhang Y, Sun YF, Wang W, Li MJ, Zhang WJ, Lin XM. Approximate nearest neighbor search on high dimensional data—Experiments, analyses, and improvement. IEEE Trans. on Knowledge and Data Engineering, 2020, 32(8): 1475–1488. [doi: 10.1109/TKDE.2019.2909204]

[30] Sun YS, Zeng JH. Research on vector database and its application. Scientific Information Research, 2024, 6(4): 11–24 (in Chinese with English abstract). [doi: 10.19809/j.cnki.kjqbyj.2024.04.002]

[31] Lu KJ, Kudo M, Xiao C, Ishikawa Y. HVS: Hierarchical graph structure based on voronoi diagrams for solving approximate nearest neighbor search. Proc. of the VLDB Endowment, 2021, 15(2): 246–258. [doi: 10.14778/3489496.3489506]

[32] Fu C, Xiang C, Wang CX, Cai D. Fast approximate nearest neighbor search with the navigating spreading-out graph. Proc. of the VLDB Endowment, 2019, 12(5): 461–474. [doi: 10.14778/3303753.3303754]

[33] Zhao X, Tian Y, Huang K, Zheng BL, Zhou XF. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. Proc. of the VLDB Endowment, 2023, 16(8): 1979–1991. [doi: 10.14778/3594512.3594527]

[34] Fu C, Cai D. EFANNA: An extremely fast approximate nearest neighbor search algorithm based on KNN graph. arXiv:1609.07228, 2016.

[35] Dasgupta S, Sinha K. Randomized partition trees for exact nearest neighbor search. In: Proc. of the 26th Annual Conf. on Learning Theory. 2013. 317–337.

[36] Zheng BL, Xi Z, Weng LG, Hung NQV, Liu H, Jensen CS. PM-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search. Proc. of the VLDB Endowment, 2020, 13(5): 643–655. [doi: 10.14778/3377369.3377374]

[37] Meng JF, Wang HY, Xu J, Ogihara M. ONe index for all kernels (ONIAK): A zero re-indexing LSH solution to ANNS-ALT (after linear transformation). Proc. of the VLDB Endowment, 2022, 15(13): 3937–3949. [doi: 10.14778/3565838.3565847]

[38] Ge TZ, He KM, Ke QF, Sun J. Optimized product quantization for approximate nearest neighbor search. In: Proc. of the 2013 IEEE Conf. on Computer Vision and Pattern Recognition. Portland: IEEE, 2013. 2946–2953. [doi: 10.1109/CVPR.2013.379]

[39] Yandex AB, Lempitsky V. Efficient indexing of billion-scale datasets of deep descriptors. In: Proc. of the 2016 IEEE Conf. on Computer Vision and Pattern Recognition. Las Vegas: IEEE, 2016. 2055–2063. [doi: 10.1109/CVPR.2016.226]

[40] Srinivasan K, Raman K, Chen JC, Bendersky M, Najork M. WIT: Wikipedia-based image text dataset for multimodal multilingual machine learning. In: Proc. of the 44th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval. ACM, 2021. 2443–2449. [doi: 10.1145/3404835.3463257]

[41] Microsoft spacev-1b. 2021. https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B

[42] Pan Y, Sun JX, Yu HF. LM-DiskANN: Low memory footprint in disk-native dynamic graph-based ann indexing. In: Proc. of the 2023 IEEE Int'l Conf. on Big Data (BigData). Sorrento: IEEE, 2023. 5987–5996. [doi: 10.1109/BigData59044.2023.10386517]

[43] Ni JK, Xu XL, Wang YX, Li C, Yao JJ, Xiao SH, Zhang XC. DiskANN++: Efficient page-based search over isomorphic mapped graph index using query-sensitivity entry vertex. arXiv:2310.00402, 2023.

[44] YouTube. 2025. https://blog.youtube/press/

[45] Li J, Liu HF, Gui CH, Chen JY, Ni ZY, Wang N, Chen Y. The design and implementation of a real time visual search system on JD e-commerce platform. In: Proc. of the 19th Int'l Middleware Conf. Industry. Rennes: ACM, 2018. 9–16. [doi: 10.1145/3284028.3284030]

[46] Xu HK, Manohar MD, Bernstein PA, Chandramouli B, Wen R, Simhadri HV. In-place updates of a graph index for streaming approximate nearest neighbor search. arXiv:2502.13826, 2025.

[47] Qader MA, Cheng SW, Hristidis V. A comparative study of secondary indexing techniques in LSM-based NoSQL databases. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 551–566. [doi: 10.1145/3183713.3196900]

[48] Wu LK, Lin WQ, Xiao XK, Xu YB. LSII: An indexing structure for exact real-time search on microblogs. In: Proc. of the 29th IEEE Int'l Conf. on Data Engineering (ICDE). Brisbane: IEEE, 2013. 482–493. [doi: 10.1109/ICDE.2013.6544849]

[49] Alsubaiee S, Behm A, Borkar V, Heilbron Z, Kim YS, Carey MJ, Dreseler M, Li C. Storage management in AsterixDB. Proc. of the VLDB Endowment, 2014, 7(10): 841–852. [doi: 10.14778/2732951.2732958]

[50] Shin J, Wang JG, Aref WG. The LSM RUM-tree: A log structured merge R-tree for update-intensive spatial workloads. In: Proc. of the

37th IEEE Int'l Conf. on Data Engineering (ICDE). Chania: IEEE, 2021. 2285–2290. [doi: 10.1109/ICDE51399.2021.00238]
[51] Xu R, Liu ZH, Hu HQ, Qian WN, Zhou AY. An efficient secondary index for spatial data based on LevelDB. In: Proc. of the 25th Int'l Conf. on Database Systems for Advanced Applications. Jeju: Springer, 2020. 750–754. [doi: 10.1007/978-3-030-59419-0_50]
[52] Gottesbüren L, Dhulipala L, Jayaram R, Lacki J. Unleashing graph partitioning for large-scale nearest neighbor search. Proc. of the VLDB Endowment, 2024, 18(1): 1649–1662.
[53] Jégou H, Tavenard R, Douze M, Amsaleg L. Datasets for approximate nearest neighbor search. 2011. http://corpus-texmex.irisa.fr/

附中文参考文献

[3] 朱俊. 向量搜索在电商商品批量检索的应用. 宝钢技术, 2023(4): 13–16. [doi: 10.3969/j.issn.1008-0716.2023.04.004]
[30] 孙雨生, 曾俊皓. 向量数据库及其应用研究. 科技情报研究, 2024, 6(4): 11–24. [doi: 10.19809/j.cnki.kjqbyj.2024.04.002]

作者简介

邱海浪, 硕士生, CCF 学生会员, 主要研究领域为向量数据库, 数据库系统, 大数据.
彭煜玮, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为数据库系统, 大数据, 数字水印.
彭智勇, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为数据管理, 数据库, 数据挖掘与分析.