Catapults to the Rescue: 利用查询局部性加速向量搜索

Sami Abuzakuk 洛桑
联邦理工学院 洛桑，瑞士
sa
mi.abuzakuk@epfl.ch

Anne-Marie Kermarrec
EPFL
洛桑 ，瑞士
anne-marie.kermarrec@epfl.ch

Rafael Pires
EPFL
洛桑 ，瑞士
rafael.pires@epfl.ch

Mathis Randl*
EPFL 洛桑，瑞士
m
athis.randl@epfl.ch

Martijn de Vos
EPFL 洛桑，瑞士
m
artijn.devos@epfl.ch

arXiv:2603.02164v1 [cs.DB] 2 Mar 2026

摘要

基于图的索引是向量数据库中进行近似最近邻搜索的主流方法，能够在数十亿个向量上实现高召回率和低延迟。然而，在此类索引中，邻近图的边集只会随着索引数据的变化而修改，从不会根据查询工作负载进行调整。这种做法效率低下：现实世界的查询流具有显著的空间和时间局部性，但每个查询仍必须从固定或随机的入口点重新遍历相同的中间跳数。我们提出了 CatapultDB，这是一种轻量级机制，首次能够动态地即时确定从近似最近邻索引中的何处开始搜索，从而利用查询局部性。CatapultDB 会添加名为 弹射器 的快捷边，将查询区域连接到经常访问的目标节点。弹射器作为图之上的附加层进行维护，因此标准向量搜索算法保持不变：如果存在合适的弹射器，查询只需被路由到更好的起始点。这种透明设计保留了底层系统的全部功能，包括过滤搜索、动态插入和磁盘驻留索引。我们实现了 CatapultDB，并使用四种偏置程度各异的工作负载对其进行评估。实验表明，与 DiskANN 相比，CatapultDB 在召回率相当或更高的情况下，可将吞吐量提升至 2.51× 倍；它的效率可媲美基于局部敏感哈希（LSH）的方法，同时无需牺牲过滤功能或重建索引；与基于缓存的替代方案不同，它还能平稳地适应工作负载变化。

1 引言

最近邻搜索（NNS）是许多领域中的关键操作，包括推荐系统 [1],大规模聚类 [20], 相似性搜索 [16], 以及近年来用于机器学习（ML）推理的检索增强生成（RAG）流水线 [18]。随着向量数据库扩展到容纳数十亿个高维数据点，精确搜索的成本变得高得令人难以承受 [17]。这促使人们采用近似最近邻（ANN）搜索：这些方法不保证返回精确结果，而是返回与查询足够接近的数据点，以少量且有限的准确度损失换取查询延迟的大幅降低 [13]。在近似最近邻搜索中，实现高吞吐量和低延迟的主流方法是构建基于图的索引 [22, 23]。这包括构建邻近图，其中向量是节点，边则连接近似最近邻。随后从指定的入口点开始，通过贪心遍历进行搜索，在图中导航，直至收敛到最接近查询向量的局部邻域。

尽管取得了这些进展，基于图的近似最近邻搜索方法仍有一个隐含限制：只有在节点集合发生变化时，邻近图的边集才会改变，例如$e_。 g_。$，即有新文档被添加到数据库中 [14, 23, 30]。绝大多数向量数据库实现都依赖于在构建时确定的边结构；一旦构建完成，该结构相对于查询工作负载便保持静态。这并非最优方案，因为真实世界的工作负载表现出很强的时空局部性：查询聚集在向量空间的某些区域，而连续的相似查询会冗余地遍历相同的探索路径 [10, 26, 28, 29]。然而，每个查询都必须从一个不考虑工作负载的入口点开始，并重新遍历冗余的中间跳数，从而为每次查找带来不必要的延迟和计算开销。

以往的工作从两个角度尝试解决这一低效问题。第一个角度是选择更好的图入口点，以缩短从入口点到查询邻域的遍历路径。例如，LSH-APG [36] 使用局部敏感哈希（LSH）[9] 来确定更好的图入口点，从而缩短遍历路径。然而，LSH-APG 的捷径仅根据数据分布在索引构建时静态计算，完全不考虑查询工作负载：其入口点索引必须与邻近图一同从头构建，因此无法叠加在现有索引之上。这也意味着 LSH-APG 无法利用运行时信息，例如基于属性的过滤谓词；这类谓词在生产环境中很常见，且只有在查询时才已知 [11]。最后，由于 LSH-APG 中的哈希函数是在构建时根据查询数据集和索引数据集中可能出现的距离范围进行校准的，任何分布外插入都会降低入口点质量，并最终需要进行完整的索引重建；对于频繁更新数据的场景而言，这种成本高得令人难以承受。

第二种思路是通过缓存先前查询的结果，更直接地利用查询局部性：如果传入的查询与过去的查询足够相似，就直接返回缓存的结果，而无需进行查找 [2, 10]。例如，Proximity [2] 引入了一种近似缓存，用于缓存查询及其结果。如果传入的查询与之前见过的某个查询之间的距离低于某个阈值，

*通讯作者

表 1：向量搜索方法的特征比较。

系统	加速搜索（局部敏感哈希）	支持插入	支持过滤	可扩展至持久化内存
LSH-APG [36]	✓	✓	✗	✗
Proximity [2]	✓	✗	✗	✓
DiskANN [14]	✗	✓	✓	✓
CatapultDB（我们的方法）	✓	✓	✓	✓

就直接返回缓存的结果，完全绕过数据库。虽然 Proximity 能有效缩短具有时间局部性的工作负载的查询查找时间，但它无法感知底层向量数据库中的动态插入；此外，它对距离阈值参数较为敏感，因此当工作负载分布发生变化时，缓存条目会变得过时，结果也会随之变差。表 1 总结了上述权衡。理想的解决方案应结合 LSH-APG 的入口质量与 Proximity 缓存的工作负载感知能力。

本文介绍了 CatapultDB，这是一种新颖的方法，首次通过动态修改邻近图的边集，利用查询工作负载的时空局部性。CatapultDB 会适应搜索轨迹中的重复模式：当向量空间某一区域（使用局部敏感哈希识别）中的查询反复从入口点遍历到同一个目标节点时，CatapultDB 会注入一条直接连接两者的快捷边，我们称之为弹射器。同一区域中的后续查询会被直接弹射到相关邻域，完全跳过冗余的中间跳数。

CatapultDB 的一项关键设计原则是透明性：它作为一个轻量级层运行于任何接受提示以指定搜索起点的索引之上，例如 DiskANN [14] 或 HNSW [23] 中的入口节点。最近邻搜索仍通过常规的贪心遍历算法进行；如果存在历史局部性信息，查询只需直接访问图中更合适的区域即可。由于弹射器是图自身中的边，而非外部缓存中的条目，CatapultDB 不会削弱底层索引提供的任何功能，包括动态插入和删除、基于属性的过滤搜索 [11], , 以及在商用硬件上扩展到十亿级数据集（这需要将部分索引存储在持久化存储中，而不是完全依赖有限的系统内存） [11, 14, 30]。

我们将 CatapultDB 实现为一个层，可叠加在任何暴露可配置入口点的基于图的近似最近邻索引之上；在评估中，我们使用了广受欢迎且部署广泛的向量数据库 DiskANN。我们使用四种局部性程度各异的工作负载评估 CatapultDB。主要实验发现如下：

(1) 通过利用工作负载局部性，大幅减少每次查询所需的图遍历路径数量，CatapultDB 在召回率相当或更高的情况下，吞吐量比原生 DiskANN 提高 2.51×，比 LSH-APG 提高 2.97×。即使在查询完全没有局部性的最差情况下，CatapultDB 通常也不会牺牲相较于基线方法的吞吐量。

（2）CatapultDB 天然支持过滤查询和动态插入，且无需索引重建。

与 DiskANN 相比，CatapultDB 的吞吐量最高可提升 1.38×，召回率最高可提高 11%。

总之，我们的主要贡献如下：

(1) 我们发现了在基于图的近似最近邻索引中在线、由查询驱动地添加边的机会，并提出了弹射器（即，捷径边）的概念：根据观察到的搜索轨迹将其注入图中。据我们所知，CatapultDB 是首个动态修改邻近图边集以利用查询工作负载局部性的系统，而非仅仅为了反映索引数据的变化（第 3 节）。

（2）我们将 CatapultDB 实现为与索引无关的层，可透明地增强任何接受搜索提示的向量索引，同时保留底层系统的全部功能，包括过滤搜索和动态插入（第 4.1 节）。

（3）我们使用四种局部性程度各异的工作负载进行了广泛实验（第 4 节）。结果表明，与 DiskANN 和 LSH-APG 相比，CatapultDB 显著提高了查询吞吐量；它原生支持过滤查询（而 LSH-APG 不支持），并能稳健地适应不断变化的工作负载，而基于缓存的方法则无法做到这一点。

2 背景与动机

我们首先介绍最近邻搜索（NNS）和基于图的近似最近邻（ANN）搜索，CatapultDB 正是构建于此基础之上；随后介绍局部敏感哈希（LSH），CatapultDB 使用它来识别查询区域，并将查询路由到索引中更合适的入口点。最后，我们通过描述在真实世界查询工作负载中观察到的局部性特征，阐明 CatapultDB 背后的关键洞见。

2.1 向量搜索

2.1.1 最近邻搜索（*NNS*）。最近邻搜索是指在大型集合 [6] 中，找出与给定查询最相似的向量这一问题。形式化地，设 D $\subset \mathbb{R}^d$ 是一个包含 n 个维度为 d 的向量的数据集。给定一个查询向量 $q \in \mathbb{R}^d$ 和一个正整数 k，k 最近邻（k-NN）查询会返回 D 中距离 q 最近的 k 个向量，所依据的是给定的距离函数（例如，欧几里得距离）。精确的k-NN 搜索需要将 q 与 D 中的每个向量进行比较；当 n 和 d 很大时，这种做法的开销极高。

2.1.2 近似最近邻（*ANN*）搜索。由于精确的 k-NN搜索无法扩展到大型数据集，近似最近邻（ANN）搜索放宽了对精确性的要求，以换取显著降低的查询延迟 [13]。一个（c, k）-ANN

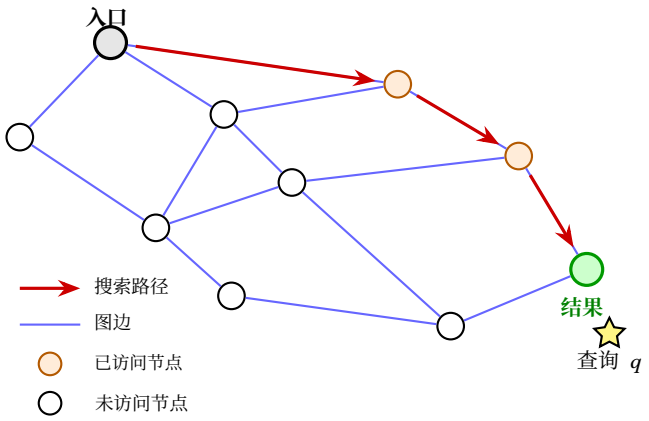

图 1：针对 $k = 1$ 近邻的标准束搜索。搜索路径采用贪心方式选择：每次跳转到与查询距离最小的近邻。

查询返回 k个向量 o_1，……，o_k使得 $\|q, o_i\| \leq c \cdot \|q, o_i^*\|$，其中 o_i^*是第 i个真实最近邻，$c > 1$ 是近似比。实践中，近似最近邻搜索方法通过召回率（即：出现在结果中的真实 k-NN所占的比例）和查询延迟进行评估。近似最近邻搜索方法主要分为三类：基于树的方法（例如KD树）、基于量化的方法（例如乘积量化 [17]）以及基于图的方法。近年来，在基准测试中，基于图的近似最近邻搜索方法一直表现出最佳的召回率与延迟权衡，并广泛应用于生产环境 [19, 34]。因此，它们构成了我们工作的基础。

2.1.3 基于图的近似最近邻搜索。基于图的近似最近邻搜索方法会构建邻近图 $G = (V, E)$，其中每个节点对应 D 中的一个向量，边则连接近似近邻。搜索通过贪心遍历进行：从一个或多个入口点出发，算法反复移动到距离查询最近的未访问邻居，并不断扩展候选集，直至收敛。算法 1 展示了基于图的索引上的这一束搜索过程。影响性能的一个关键参数是起始点集合 sp：它们的质量直接决定了到达查询邻域所需的跳数。这种方法要求图具有一种结构，使得这种贪心搜索能够找到合理的结果；这种性质称为可导航性 [4]。

2.1.4 过滤近似最近邻。许多实际应用会为向量附加元数据标签，并要求查询结果满足关于这些标签的谓词，例如，“在所有符合谓词 f 的向量中，返回 k 个最近邻 q”，其中 f 是文档 [11] 关联标签的布尔函数。形式上，一个过滤（c, k）近似最近邻查询会返回 k 个向量 o_1，……，$o_k \in \mathrm{D}f = \{x \in \mathrm{D}: f \in F_x\}$，使得 $\|q, o_i\| \leq c \cdot \|q, o_i^*\|$，其中 o_i^* 是 Df 内第 i 个真实最近邻。一个简单的方法是对未过滤查询的结果进行后置过滤，但对于低特异性标签（稀有过滤条件），这种方法无法保证正确性，因为未过滤搜索可能返回零个符合谓词的点。FilteredVamana [11] 通过将标签信息直接纳入图构建和搜索来解决这一问题：每个查询都从专属的、按标签划分的入口点开始，并扩展剪枝规则以保留

算法 1： 基于图的束搜索（改编自 DiskANN [14]）

输入： 查询向量 q，邻居检索数量 k，初始起始点 sp

过程 查找(q, k, sp)：
- $C = sp$ //候选项集合
- $V = \{\}$ //已访问节点集合
- **当** $C \setminus V \neq \{\}$ **执行**
  - $currentCandidate = x \in C \setminus V$最接近 q
  - $V = V \cup \{currentCandidate\}$
  - $C = C \cup$近邻$(currentCandidate)$
  - $C = \mathrm{trim}(C, k, q)$ //最多保留 k个最接近 q的
- **return** C

过滤子图可导航性。这确保贪心遍历始终处于图的相关子集中，即使过滤条件覆盖的数据集不到 1%，也能保持较高的召回率。包括 Pinecone [27]、Weaviate [35]，和 Milvus [33]，在内的主流生产级向量数据库，都将过滤搜索作为一项一等功能提供。因此，支持过滤搜索是当代任何向量数据库的一项重要要求。

2.2 局部敏感哈希（LSH）

加速基于图的近似最近邻搜索的一种方法是利用局部敏感哈希（LSH）确定进入该图的更好入口点，从而减少到达查询邻域所需的跳数。LSH [9]是一族哈希函数，其设计使得原始空间中邻近的点比远处的点更可能发生碰撞（即共享相同的哈希值）。形式上，函数族 H $= \{h: \mathbb{R}^d \to \mathbb{R}\}$ 是 (r, cr, p_1, p_2)-局部敏感的，若对任意两点 o_1、o_2：(i)$\|o_1$、$o_2\| \leq r$蕴含 $\Pr[h(o_1) = h(o_2)] \geq p_1$，且 (ii) $\|o_1$、$o_2\| > cr$ 蕴含 $\Pr[h(o_1) = h(o_2)] \leq p_2$，并满足 $p_1 > p_2$。

在本工作中，我们使用随机超平面 局部敏感哈希将查询空间划分为若干区域（见第 3 节）。在该局部敏感哈希实例化中，每个哈希函数由一个从标准正态分布中抽取的随机向量 $\vec{r} \in \mathbb{R}^d$定义。该向量表示一条穿过原点的 d维超平面的法向量。点 o被映射为单个比特：$h(o) = 1[\vec{r} \cdot \vec{o} \geq 0]$，表示 o所在的超平面一侧（哈希值为 1 表示与法向量 r同侧）。通过拼接 L 个这样的比特，可以得到一个 L位哈希码，从而将空间划分为2^L个桶。我们注意到，Proximity [2] 也使用该变体来索引其缓存。

2.3 真实世界查询工作负载中的时间局部性与空间局部性

现实世界中的查询工作负载呈现出高度的时空局部性。从词汇层面来看，众所周知，大规模搜索日志中的查询频率遵循重尾、近似幂律（齐普夫）分布：少数查询被极其频繁地提交，而绝大多数查询则很少出现 [26, 29]。这种结构性偏斜在不同领域和规模中都得到了一致观察，从通用

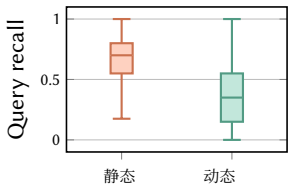

图 2：Proximity 在处理静态向量数据库（左）与动态向量数据库（右）时的查询召回率分布。

网络搜索日志 [29] 到健康领域检索平台皆是如此。例如，在 TripClick [28], 中，一个从医学搜索引擎收集的、涵盖七年间 520 万次用户交互的数据集，其查询频率直方图呈指数趋势：少数查询（如 “妊娠期哮喘”）占据了不成比例的流量份额，而长尾部分则由仅提交过几次的查询组成。

在任何合理的嵌入模型下，语义相似的查询都会映射到彼此接近的向量，因此查询日志中的词汇偏斜会直接转化为向量空间中的空间偏斜。当用户查询被编码为稠密向量并提交到向量数据库时，生成的查询工作负载会集中在高维空间中的少数几个区域。多项针对工业级向量搜索工作负载的研究证实，对索引的访问模式高度不均匀 [10, 24, 31]。一项关于知识图谱实体搜索工作负载的研究发现，在全天的流量中，仅约 15% 的倒排文件（IVF）索引被访问过 [24]。这种访问偏斜会对系统设计产生实际影响：QVCache [10]表明，兆字节级的缓存就足以处理针对十亿点索引的大量查询，恰恰是因为活跃工作集相对于整个数据集而言很小。

除了空间上的集中性，工作负载还表现出很强的时间局部性：连续的查询往往会在嵌入空间中聚集，语义相似的查询会在短时间窗口内成批到达 [5]。例如，这种行为自然出现在对话式应用中，用户会在单个会话中提出相似且相关的问题；也会出现在电子商务场景中，多个用户同时搜索同一款热门商品（例如，夏季的空调） [3]。然而，由于连续的查询从来不是完全重复的，每个查询都会被视为一次独立查找，导致工作负载的空间和时间结构完全未得到利用。

一些系统（例如 Proximity）通过在向量数据库与应用层之间部署近似缓存作为中间件，来利用查询的局部性。它会观察流量，包介于两者之间，并拦截与缓存中已存储的先前查询足够相似的查询。随后，它会立即返回对应的已存储文档。这类系统无法容忍底层数据库中的插入和删除操作：它们要么在数据库每次更新时删除自身状态，从而失去对所有累积历史的跟踪；要么提供过时的结果。因此，对于任何正在进行数据摄取的向量数据库而言，这类系统都不适用。我们通过实验对此进行了证明：首先使用 PubMed

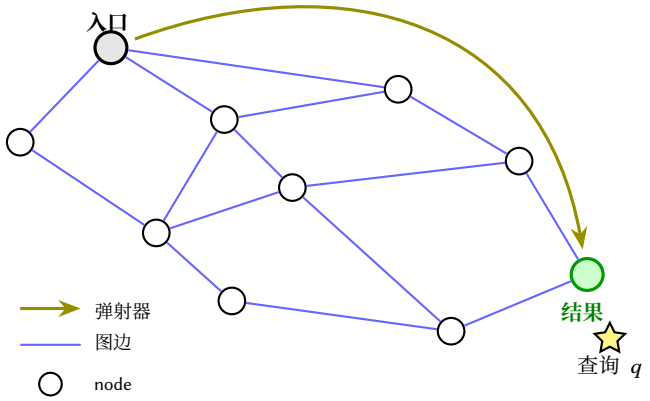

图 3：由弹射器加速的邻居束搜索 $k = 1$。查询不会从入口点开始遍历多个中间跳数，而是通过弹射器边直接进入目标邻域，从而显著缩短搜索路径。

数据集生成的嵌入向量填充向量数据库，然后将 MedRAG-Zipf 查询工作负载（第 4.1 节将进一步介绍）发送至 Proximity。在一次实验运行中，数据库中存储的向量保持静态；而在另一次运行中，动态插入新的向量（每 50 个查询插入一批包含 5000 个向量的数据），导致 Proximity 提供过时的结果。图 2 展示了静态运行和动态运行的召回率。当向量数据库中的内容发生变化时，召回率会大幅下降：在静态向量数据库中，召回率中位数为 70%，而在动态向量数据库中则降至 35%。

3 CatapultDB 的设计

受到真实世界查询工作负载局部性特性的启发，我们围绕四个目标设计了 CatapultDB：（*i*）工作负载感知，即，利用查询流中的空间和时间局部性来减少冗余图遍历；（*ii*）支持动态更新，在插入或删除向量时仍保持高效，无需重建索引；（*iii*）支持过滤查询；以及（*iv*）与索引无关，作为轻量级层运行于任何公开可配置入口点的现有基于图的索引之上，且无需更改底层搜索算法。

我们首先在第 3.1 节阐释 CatapultDB 背后的总体思路和洞见，然后在后续小节中详细介绍 CatapultDB 的各个组件。

3.1 CatapultDB 概述

由于现实世界中的查询在空间上集中、在时间上相关，因此，从向量空间某个区域近期查询中收集的信息，能够很好地预测未来该区域中的查询将遵循哪些遍历路径。这正是 CatapultDB 的核心洞见：CatapultDB 不会在查询完成后丢弃每次遍历的结果，而是将其记录下来，用于缩短同一区域内未来查询的起始路径，直接利用静态索引所忽略的工作负载局部性。我们将这些捷径称为弹射器：它们是有向边，

算法 2： 在现有搜索之上构建弹射层

输入： 查询向量 q，邻居检索数量 k，图中位点 m，初始为空 $buckets$，桶容量 b

```
1 过程 弹射器化_查找(q, k, m, lsh, buckets):
2     hash = 局部敏感哈希(q)
3     catapults = buckets[hash]
4     sp = catapults ∪ {m}                    //起始点
5     ann = 查找(q, k, sp)                    //参见算法 1
6     catapults.append(ann.best())
7     if |catapults| > b then
8         catapults.lru_驱逐()
9     return 近似最近邻
```

将一个查询区域（由其 LSH哈希码标识）连接到邻近图中更理想的入口点。图 3 展示了这一过程。

我们将 CatapultDB 设计为一个轻量级层，根据观测到的查询工作负载动态调整底层基于图的近似最近邻索引的入口点。弹射器作为辅助层维护在索引之上，并作为额外的起始点传递给标准近似最近邻过程（*e.g.*, 算法 1）。底层搜索算法完全保持不变。

3.2 弹射器机制

接下来，我们将详细介绍弹射器子系统。

弹射式查找。算法2展示了弹射式_查找过程，该过程用于通过 CatapultDB 执行近似最近邻搜索。此算法调用算法 1 中的查找过程，该过程由底层数据库提供，但它会创建并管理弹射器，以加速向量搜索。弹射式_查找过程的输入包括查询向量 q、希望检索的邻居数量 k、图中位点 m，以及一个包含弹射器的桶列表。图中位点是与所有其他点的平均距离最近的向量；将其纳入考虑，是为了应对弹射器集合为空或与查询无关的情况。CatapultDB 使用局部敏感哈希将搜索入口点组织到桶中，本节后文将对此进行更详细的讨论。

给定一个输入查询 q，首先计算其 LSH 哈希码，并检索对应的桶（第 2–3 行）。桶中的内容与数据集的中心点 m 共同构成起始点集合 sp，并将其传递给底层的束搜索（第 4 行）。查找过程完成后，会返回一个列表 ann，其中包含至多 k 个被认为与 q接近的向量。随后，将 ann 中的最佳结果（即邻近图中与 q 最接近的返回邻居节点）添加到该桶的弹射器列表中（第 6 行）。此节点将作为弹射器的目的地，供之后映射到同一 LSH 区域的查询使用，从而有效地创建一个新的弹射器。后续相似查询无需从中心点 m 开始遍历并在邻近图中经过多个跳数，而是会被直接弹射到相关邻域，从而利用空间局部性。值得注意的是，将弹射器作为入口点注入，等同于在 LSH 匹配时添加一条从中心点指向弹射器着陆节点的边。我们的方法不过，由于弹射器机制与束搜索（算法 1）的交互仅限于起始点接口，因此我们可以将二者解耦。

请注意，如果 CatapultDB 从未见过查询 q，但该查询映射到了一个非空的局部敏感哈希桶，那么内部搜索方法所使用的入口点仍可能意外地发挥作用：该桶中存储的弹射器落点来自同一局部敏感哈希区域内过去的查询记录，因此比图中位点更有可能位于 q 的邻域附近。具体而言，图中位点的选择旨在通过贪心导航，尽可能减少到达任意点所需的跳数，而不考虑查询分布；CatapultDB 则利用了查询分布。

基于 *LSH* 的区域识别。 由于基于图的近似最近邻索引通常从邻近图中的单个固定入口点启动每个查询，若简单地从该节点向图中所有目标区域添加弹射边，会使该节点因大量出边而过载。所有这些边都必须在执行第一跳之前进行评估，从而大幅增加初始搜索开销。

为显著降低处理查询的计算成本，我们使用局部敏感哈希来计算当前查询与生成该弹射器的前一个查询之间的相似度。在实践中，CatapultDB 使用随机超平面局部敏感哈希将查询空间划分为多个区域。给定传入的查询向量$q \in \mathbb{R}^d$ 以及 L 个随机生成的、法向量为r_1、…、r_L的超平面，q的LSH哈希码为（$q \cdot r_1 \geq 0$，…，$q \cdot r_L \geq 0$），如第2.2节所述。共享同一哈希码的查询被视为属于同一区域，并由同一个弹射桶处理。由于随机超平面局部敏感哈希是尺度不变的，因此无需进行数据集特定校准，这与 LSH-APG 所使用的 p-稳定局部敏感哈希变体形成对比。

***Catapult* 淘汰。** 每个桶都分配有固定容量，该容量定义为全局预设常量 b（见第 3.3 节）。当桶已满且要插入新的弹射器时，将根据最近最少使用（LRU）淘汰策略淘汰一个现有入口（算法 2 的第 7–8 行）。这种淘汰机制对于 CatapultDB 适应工作负载变化至关重要：淘汰最近最少使用的入口，可以确保与较早、可能已过时的查询模式相关联的目标节点被当前工作负载所需的目标节点替换。这种设计还使系统能够适应底层图中的文档插入：随着新向量被添加到索引中，它们可能比桶中当前存储的目标节点更适合作为弹射器目标节点。由于最近发出的查询在遍历过程中自然会到达这些新节点，相应的桶入口会随时间推移而更新；指向先前最优、但现已被取代的节点的过时目标节点也会逐渐被淘汰。无需显式通知索引更新；淘汰机制会通过正常的查询流被动适应。

邻近图构建。 CatapultDB 作为现有邻近图之上的一层运行，无需从头构建图。在评估 CatapultDB 时（见第 4 节），我们使用 Vamana，即 DiskANN 所采用的图构建算法。Vamana 在实测中表现出色，而且

在可能采用 CatapultDB 的部署环境中，很可能已经可用。

竞争性召回。 基于图的近似最近邻索引旨在具备可导航性，*i*。*e*。，也就是说可以通过贪心方式遍历。这意味着入口点的选择只会影响到达查询邻域所需的跳数，而不会影响最终能否找到该邻域。因此，召回率不受起始点质量的影响：较差的弹射器只意味着遍历从距离目标更远的位置开始，需要更多跳数，可能会降低吞吐量，但绝不会影响正确性。将图中位点纳入起始点集（算法 2 第 4 行）进一步强化了这一特性。即使某个 LSH 区域没有记录任何弹射器，中位点也能提供与未经修改的 DiskANN 搜索相同的基线。弹射器旨在带来非负收益：若能提供有效信息，就能提升吞吐量；若不能，则理想情况下会退回基线行为。实践中，在查询工作负载缺乏局部性的罕见情况下，吞吐量可能会略有退化（见第 4.3 节）。

存储成本可忽略不计。弹射器目标节点以节点标识符的形式存储在邻近图中，几乎不会带来额外内存开销：大小为 b 的桶在每个 LSH 桶中仅需存储 b 4 字节的索引，所有 2^L 个桶总计需要 $b{\cdot}2^L$ 个整数。按照本文中 $b = 40$ 和 $L = 8$ 的典型取值，CatapultDB 的额外内存需求仅为 40 KiB 的边数据，超出底层束搜索算法本身的内存需求。在实践中，这轻松满足常规部署的内存预算。相比之下，图向量索引的大小通常可达数十或数百 GB，其中大部分空间用于存储全精度向量和压缩向量 [25]。

与 *DiskANN* 集成。DiskANN [14] 将邻近图存储在固态硬盘上，同时仅在内存中保留压缩向量表示，因此能够在商用硬件上扩展至十亿级数据集。由于弹射器子系统只提供入口点，并将搜索委托给底层的束搜索实现，CatapultDB 自然继承了 DiskANN 的可扩展性。

由于每次查询的跳数决定了固态硬盘读取次数，因此减少跳数可直接降低 I/O 延迟。弹射器还能进一步放大这一效果：通过从更接近目标邻域的位置启动搜索，CatapultDB 减少了跳数，进而减少磁盘读取次数，从而进一步降低 DiskANN 的延迟。

3.3 CatapultDB 参数

CatapultDB 引入了两个系统参数：LSH 超平面的数量 L 和桶容量 b。我们将讨论这两个参数对 CatapultDB 性能的影响。

LSH 超平面的数量（L）。 L 决定了 CatapultDB 划分查询空间的粒度。每增加一个超平面，桶的数量就会翻倍，总共形成 2^L 个区域。 L 的值越大，区域的粒度就越细，从而降低两个不同查询落入同一个桶并被分配到相同弹射器目标节点的可能性。然而，增大 L 也会使 2^L 中的最大内存消耗线性增长，因为每增加一个超平面，桶的数量就会翻倍；同时还可能使各个桶过于细分，导致它们很少被重复使用，从而削弱 CatapultDB 所利用的查询局部性。较小的 L 会形成更粗粒度的区域，带来更高的复用率，但弹射器目标节点也可能因此包含更多噪声。一般而言， L 应经过调优，使每个桶中的预期查询数量足够多，以便快速填充弹射器，同时又要足够少，从而确保各个桶对应于查询空间中具有实际意义且连贯的区域。我们的实证结果表明，在各项实验中，$L = 8$能带来最高的端到端吞吐量（见第 4.5 节）。

桶容量（b）。 b 表示每个 LSH 桶中保留的弹射器目标的最大数量，并决定了自适应性与稳定性之间的权衡。较小的 b 使 CatapultDB 能够快速响应工作负载变化，即，随着新查询到来，过时的目标会迅速被逐出；但当同一区域内的查询各不相同时，也可能丢弃有用的弹射器。较大的 b 会在桶中积累更多历史信息，并在工作负载稳定时获得更高的命中率，但代价是每次查询需要评估更多候选起始点。通过实证，我们发现 $b = 40$ 在我们的实验中实现了最高的端到端吞吐量（见第 4.5 节）。

底层图属性。 除 CatapultDB 自身的参数外，底层图索引的属性也会影响弹射器能够带来的收益。接下来，我们将讨论 Vamana 构建的图中的重要参数，因为我们主要使用 DiskANN 来评估 CatapultDB 的性能。底层 Vamana 图有三个关键参数：最大出度 R、剪枝参数 α 和数据集大小 N。这三个参数是 Vamana 图的属性，并非由 CatapultDB 自身设置，但它们会影响弹射器能够带来的收益。最大出度 R 控制每跳评估的邻居数量：度更高的图每一步会探索更多邻域，但由于需要评估更多邻居，每一步耗时也会增加。 $\alpha > 1$ 是 Vamana 的剪枝参数，它以适度增加度为代价，大幅缩小图直径，从而减少从中心点到达任意查询邻域所需的跳数。使用较大的 α 构建的图，其遍历路径本已较短，因此弹射器能够节省的绝对开销较小；相反，较低-α 的图基线遍历路径更长，因而有更多空间通过弹射器缩短遍历。最后，数据集大小 N 会影响 LSH 桶覆盖范围相对于图的粗细程度：随着数据集增大，2^L 桶覆盖的向量空间区域也按比例变大，这可能会降低弹射器目标节点的精度，因此提示应相应增大 L。

3.4 过滤支持

如第 2.3 节所述，向量搜索工作负载会将元数据标签与已索引的文档关联起来，并要求查询结果满足针对这些标签的谓词。FilteredVamana [11]通过将标签信息纳入图的构建和搜索来解决这一问题：为每个查询分配一个按标签入口点，

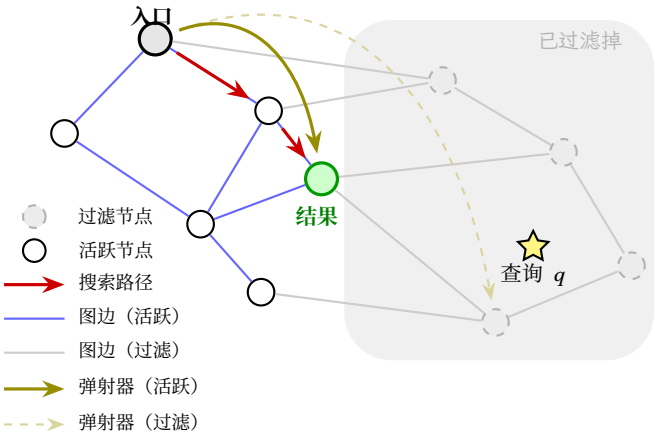

图 4：在 CatapultDB 中处理过滤查询。运行时会将过滤器关联到查询，从而禁用邻近图的部分区域。弹射器可以有条件地禁用。

并将贪心遍历限制在满足活跃过滤器的节点范围内。CatapultDB 可以自然地与这一机制结合。

收到过滤查询时，照常根据查询向量计算其 LSH 哈希码。不过，在将对应桶中存储的弹射器目标节点加入起始点集之前，还会根据活跃过滤谓词对其进行检查。不满足该谓词的目标节点会从 sp 中排除；对于该起始位置，搜索会回退到 FilteredVamana 提供的按标签入口点。图 4 也展示了这一过程。这确保贪心遍历始终处于过滤器定义子图内，即使弹射器处于启用状态，也能保持 FilteredVamana 的正确性保证。请注意，这种系统无法在索引构建时实例化。可能的查询过滤器空间远大于文档标签集合，因此无法实际为每种可能的过滤器组合预先计算并存储有意义的入口点。

过滤查询完成后，在返回结果中排名最高且必然满足过滤谓词的邻居会被插入桶中，并标记为活跃过滤器。因此，后续具有相同 LSH 哈希码和过滤谓词的查询会被直接弹射到相应的过滤后邻域，从而进一步提升 CatapultDB 和 FilteredVamana 所实现的延迟降低效果。

3.5 多线程与并发

生产环境中的向量数据库会并发处理查询，将每个传入请求分派到单独的 CPU核心。所有 CPU核心共享同一个内存图实例。CatapultDB 沿用这一模型：各个独立查询并行执行，每个查询都在自己的线程上运行算法 2 中的 Catapulted_Lookup 过程，而底层向量数据库无需进行任何协调。

并发带来了一个挑战，这一挑战在表 1 所比较的系统中是 CatapultDB 独有的。DiskANN 和 LSH-APG 在查询时将其索引结构视为只读，而 Proximity 则采用单线程执行。相比之下，弹射器会在查询执行期间动态更新。因此，CatapultDB 必须同步对弹射数据结构的访问，以避免多线程执行时出现未定义行为。除了正确性之外，也应考虑活性。如第 2 节所述，查询具有很强的局部性。更具体地说，空间上相似的查询可能会在短时间内成批到达。为了利用这一特性，已完成的查询必须以一种能让其他核心上并发启动的查询立即看到其弹射器的方式发布它，从而使整批查询都能受益于这一捷径。

我们使用与每个桶关联的读写锁来满足这两项要求。Catapulted_Lookup 开始时，线程以读模式获取相应桶的锁（第 3 行），允许多个查询并行读取同一个桶，同时阻止并发写入。随后，线程读取桶的内容并释放锁。底层搜索完成且新的弹射目标准备好发布后（第 6–8 行），线程会以独占写模式重新获取同一个桶的锁。这可能需要等待正在进行的读或写操作所在线程释放锁；不过，需要注意的是，处理映射到不同桶的查询的线程不会相互干扰。随后，线程追加新的目标；如有必要，则逐出最近最少使用的入口，然后再次释放锁。由于这两个临界区都很短，实际中的锁竞争可以忽略不计。我们在第 4 节通过实验对此进行了验证：锁竞争不会引入额外开销，也不会对性能造成可测量的影响；在所评估的工作负载中，端到端吞吐量会随核心数量线性增长。

4 评估

我们评估 CatapultDB 和基线方法的性能，并回答以下问题：

(1) 在使用有偏和无偏查询工作负载时，改变束宽和使用的线程数，CatapultDB 与基线方法（DiskANN 和 LSH-APG）的吞吐量（以每秒查询数计）是多少（第 4.2 节和第 4.3 节）？(2) 使用过滤查询时，CatapultDB 与 DiskANN 相比性能如何（第 4.4 节）？(3) CatapultDB 的超参数对可实现吞吐量有何影响（第 4.5 节）？

4.1 实验设置

4.1.1 工作负载。我们使用四种查询工作负载与数据集的组合来评估 CatapultDB 和基线方法，即填充向量数据库所用的数据。

TripClick [28]是一个大规模医疗领域搜索引擎日志，包含针对由 3500 万篇 PubMed 文档组成的语料库发出的 530 万次查询。我们使用 MedCPT [15] 将查询和文档都嵌入为 768 维向量，并将文档生成的嵌入向量存入向量数据库。我们按顺序重放查询，以保留真实用户流量固有的时间局部性。这个具有真实世界时间局部性的工作负载是我们的主要基准测试。

MedRAG-Zipf 与 TripClick 使用相同的文档语料库。不过，我们通过提示大语言模型 (LLM) 生成一组基础医学问题的语义等价改写句，从而构建不同的查询工作负载，并在嵌入空间中形成近重复查询簇。为构建查询工作负载，我们按照参数为 0.8 的齐夫分布对这些查询簇进行采样，以模拟真实搜索日志中典型的重尾查询频率偏斜。由于查询分布由人工生成问题的嵌入向量构成，因此生成的工作负载是半合成的。

Uniform 由从 $[-1,1]^d$ 中均匀随机抽取的 150 000 个查询向量组成。我们使用 PubMed 语料库中的前 100 万个文档的 MedCPT 嵌入向量来填充向量数据库。这一完全合成的查询工作负载旨在分析 CatapultDB 的最坏情况：不存在空间或时间局性的场景。借此，我们可以评估在没有局部性可加以利用时，与 DiskANN 相比，CatapultDB 是否会导致任何吞吐量退化。

Papers 是一个自然数据集，包含针对 arXiv 上发布的 270 万篇学术论文发出的 10 000 个查询。文档表示为从论文摘要中提取的 4096 维嵌入向量，并且每个文档都标注有其主要 arXiv 类别。在我们的过滤搜索实验中，这些类别标签被用作过滤谓词（见第 4.4 节）。

4.1.2 基线方法。 我们将 CatapultDB 与两种基线方法进行比较：DiskANN 和 LSH-APG。

DiskANN [14] 是一种最先进的向量数据库。DiskANN 将其邻近图存储在持久化 SSD 内存中，而压缩向量（通过乘积量化）则保存在内存中，以便在遍历期间进行快速的近似距离比较；完整向量仅针对最终候选项从磁盘读取。该设计可在商用硬件上扩展至数十亿个向量。我们将 CatapultDB 的性能与 DiskANN 的基础实现进行比较。搜索过程如算法 1 所示，并以数据集的中心点作为默认起始点。

LSH-APG [36] 用轻量级的基于 LSH 的索引增强邻近图，从而为图遍历提供更好的入口点。其核心思想是，并非每次查询都从图中位点开始，而是先通过哈希识别向量空间中靠近查询的区域，再从已接近该区域的节点启动图遍历。在构建索引时，每个被索引的向量都会映射为一个 Z 序哈希值，这些值存储在排序数值索引中。对该索引进行二分搜索可快速检索一组邻近候选项，并将其复用为起始点，从而缩短搜索路径。

然而，与 CatapultDB 相比，LSH-APG 存在实际局限。它的 LSH 索引必须与邻近图一同从头构建，因此无法叠加在 DiskANN 等现有索引之上。此外，哈希函数必须预先针对数据集的规模进行校准，这限制了它在新向量不断到达的流式场景中的适用性。最后，由于 LSH 结构在索引构建时创建，且不感知查询时谓词，LSH-APG 不支持过滤搜索。尽管存在这些局限，LSH-APG 仍是一个有力的基线：就理念而言，它是与 CatapultDB 最接近的先前工作。

4.1.3 硬件与实现。所有实验均在部署于洛桑联邦理工学院 RCP 集群上的 Docker 容器中进行，集群配备英特尔至强金牌 6240 处理器，使用的系统内存最高可达 350GB。默认情况下，实验使用统一的 Rust 代码库实现 DiskANN 式束搜索、CatapultDB 和 LSH-APG，从而确保各系统之间的比较公平。启用过滤查询的实验（参见第 4.4 节）是在 DiskANN 的定制 C++ 分支上进行的，该分支经过扩展，可在过滤场景中支持弹射器机制。两种实现均采用查询级并行和 AVX SIMD 指令进行距离计算。

4.1.4 指标。我们从两个主要方面评估系统性能：吞吐量和召回率。吞吐量以每秒查询数（QPS）衡量，并作为我们的主要性能指标。我们调整两个直接影响吞吐量—召回率权衡的关键参数。束宽 k 决定每个查询检索多少个最近邻，以及遍历过程中保留多少个候选项：值越大，召回率越高，但每个查询的计算开销也越大。线程数 t 反映系统可用的查询级并行程度。在生产部署中，多个查询会同时得到处理，因此这是主要的扩展机制。召回率衡量近似搜索返回的真实 k 个最近邻所占比例。我们同时报告召回率和吞吐量，因为只有在不同系统的召回率保持一致时，吞吐量的提升才有意义。所有实验结果均为连续三次随机种子运行结果的平均值。

4.2 CatapultDB 与基线方法在偏置工作负载下的性能

现在，我们将量化 CatapultDB 和基线系统在 MedRAG-Zipf 与 TripClick 工作负载下的性能。

4.2.1 吞吐量和召回率。图 5(a-c) 展示了 CatapultDB、DiskANN 和 LSH-APG 在 MedRAG-Zipf 工作负载下，改变束宽 k 和线程数 t 时的吞吐量（以每秒查询数计）。与基线方法相比，CatapultDB 始终能实现更高的吞吐量。对于 $t = 4$ 和 $k = 1$，CatapultDB 的吞吐量为 27 078.6 QPS，而 DiskANN 和 LSH-APG 分别为 12 154.5 和 11 485.1 QPS。增加 t 时，我们观察到 DiskANN 和 LSH-APG 的吞吐量在 k 的不同取值下基本相当；LSH-APG 的表现略逊于 DiskANN，因为其 LSH 索引在遍历过程中会引入开销。然而，CatapultDB 的吞吐量显著提升：从 $t = 4$ 和 $k = 1$ 时的 27 078.6 QPS，提升到 $k = 1$ 和 $t = 16$ 时的 38 333.2 QPS。随着我们增大 k 并保持 t 不变，还观察到 CatapultDB 相对于基线系统的吞吐量优势有所下降。对于 $k = 16$ 和 $t = 4$，CatapultDB 的吞吐量为 15 328.0，而次优基线 DiskANN 为 7623.9。这一趋势可由弹射器机制的性质解释：无论 k 如何，找到一个合适候选起点所需的工作量都是恒定的，而扩展该候选点周围的邻域所需的工作量则随 k 增加。弹射器只能加速前者，因此随着 k 增大，它们对端到端查询时间的相对贡献会逐渐减小。图 5(d) 展示了 CatapultDB 和

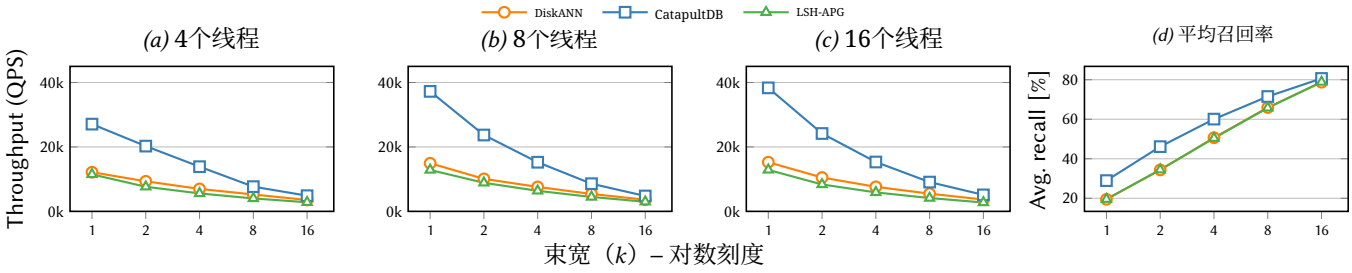

图 5：CatapultDB 和基线方法的吞吐量（每秒查询数）与平均召回率（右图，四个线程），其中束宽和线程数不断变化。我们使用 MedRAG-Zipf 工作负载。

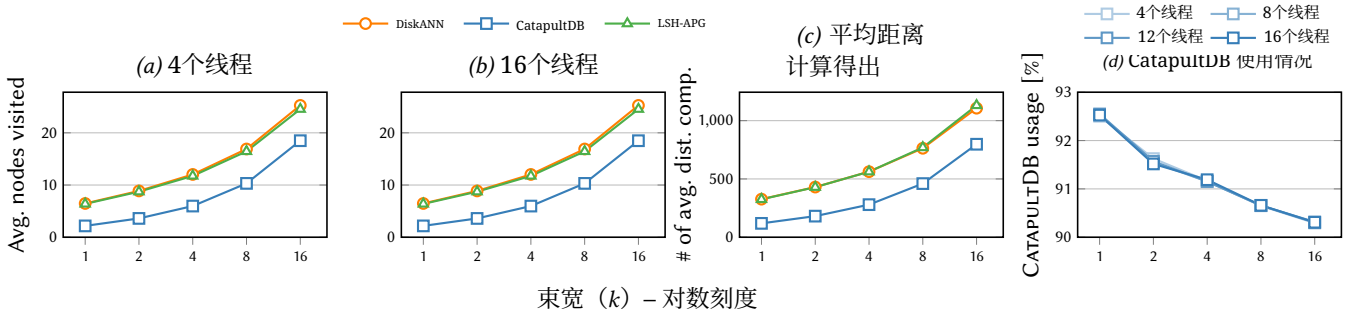

图 6：（a、b）CatapultDB 和基线方法在邻近图中访问的平均节点数，其中束宽和线程数不断变化。（c）CatapultDB 和基线方法计算的平均距离数，其中束宽不断变化 k。（d）查找期间使用弹射器的查询比例，其中束宽和线程数不断变化。我们使用 MedRAG-Zipf 工作负载。

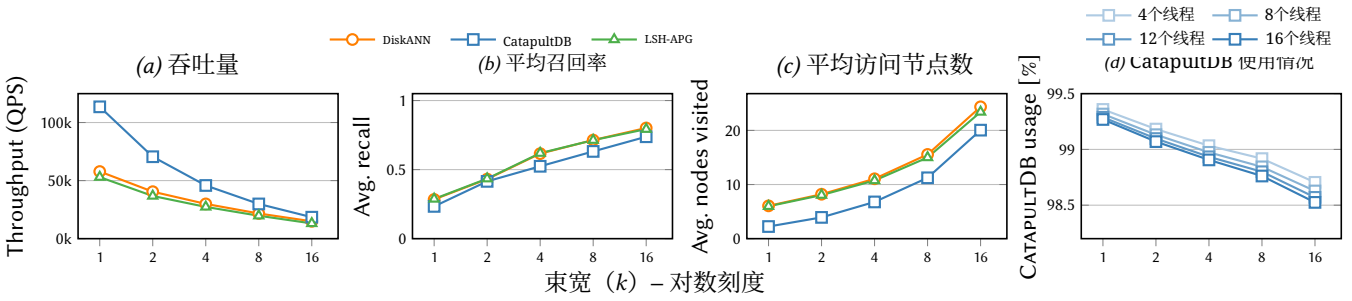

图 7：对于不同的束宽 k 和线程数，CatapultDB 与基线系统的（a）吞吐量（每秒查询数）、（b）平均召回率（四个线程）、（c）平均访问节点数和（d）CatapultDB 使用情况。我们使用 TripClick 工作负载。

基线方法的平均召回率。CatapultDB 始终能实现更高的召回率，因此在确定最近邻方面比 DiskANN 和 LSH-APG 更有效。随着 k 增大，召回率提升幅度会减小：与 DiskANN 相比，CatapultDB 在 $k = 1$ 时将召回率提高了 48.6%，而在 $k = 16$时提高了 2.4%。因此，图 5 展示了 CatapultDB 的效率：其每秒查询数最高可达基线方法的 2.51× 倍，同时召回率也优于或不低于基线方法。

4.2.2 计算开销。 图 6(a,b) 展示了在查找过程中，CatapultDB 与基线方法在邻近图中的平均访问节点数，同时变化束宽 k，以及针对 $t = 4$和 $t = 16$。DiskANN 与 LSH-APG 在每次查找中访问的节点数相当。然而，CatapultDB 在图中始终访问更少的节点，因此能够更快到达目标节点。这种减少在 k取值较低时更为显著。对于 $t = 4$ 和 $k = 1$，与 DiskANN 相比，CatapultDB 的访问节点数减少了 66.3%，而对于 $t = 4$ 和 $k = 16$则减少了 26.9%。对所有系统而言，变化 t 对访问节点数没有显著影响。图 6(c) 展示了平均距离

在变化束宽 k 的情况下，各评估系统在一次查找中所计算的。与 DiskANN 相比，CatapultDB 在 $k = 1$ 和 $k = 16$ 下分别将距离计算次数减少了 63.5% 和 27.9%。值得注意的是，在所有设置下，访问节点数与距离计算的减少幅度彼此高度吻合，这证实 CatapultDB 的收益完全来自对图遍历的捷径化，而非任何次要效应，并且弹射器查找本身引入的开销可以忽略不计。

4.2.3 弹射器使用情况。 最后，我们在图 6（右）中展示了针对不同 t 值，在查找期间使用至少一个弹射器的查询所占的百分比。CatapultDB 的使用率很高，例如，对于 $k = 1$ 和 $t = 4$，使用率达到 92.5%，凸显了我们的方法在加速查询方面的有效性。随着 k 增大，使用率略有下降，因为更宽的束对起始点不那么敏感，因此在其遍历路径中受益于弹射器的比例更小。值得注意的是，改变 t 不会影响使用率，这是一个积极的结果：尽管共享桶中存在并发写入，读写锁方案并未引入可测量的陈旧性或争用，从而验证了 CatapultDB 的并发设计。

4.2.4 在*TripClick*上的性能。 图 7 展示了 CatapultDB 和基线方法在 TripClick 工作负载下的吞吐量、召回率、访问节点数和弹射器使用情况。这些趋势与在 MedRAG-Zipf 上观察到的趋势非常相似。在所有 k 和 t 值下，CatapultDB 的吞吐量始终高于 DiskANN 和 LSH-APG（最高提升 89.7%），但召回率略低。CatapultDB 的平均访问节点数最多比基线方法低 62.7%，这证实弹射器能够有效缩短遍历路径。最后，我们观察到，在所有配置下，弹射器使用率都保持在 98% 以上，这表明 CatapultDB 利用了 TripClick 查询工作负载中固有的高度时空局部性。

4.3 CatapultDB 与基线方法在无偏工作负载下的性能

接下来，我们评估 CatapultDB 和基线方法在均匀无偏工作负载下的性能。这代表了一种最坏情况：查询从弹射器中获益的可能性最低。

4.3.1 吞吐量。图 8 展示了 CatapultDB、DiskANN 和 LSH-APG 的吞吐量（每秒查询数），并对比了不同束宽 k 和线程数 t下的表现。当 $k \geq 4$时，CatapultDB 的吞吐量与基线方法相当，表明在这些设置下 CatapultDB 不会牺牲吞吐量。尽管如此，我们注意到 $k = 1$时性能有所下降。当 $t = 4$时，CatapultDB 在 $k = 1$ 下的吞吐量比表现最佳的基线方法 DiskANN 低 25.4%。随着 t增加，这种下降变得不那么明显：对于 $t = 16$，CatapultDB 在 $k = 1$ 下的吞吐量比 DiskANN 低 20.6%。然而，我们认为，实际应用中的许多近似最近邻系统会在遍历过程中多取候选项以提高召回率，然后从返回列表中选出前 k 个候选项。因此，在实践中，CatapultDB 不会牺牲吞吐量，同时还能在查询工作负载具有时间偏差时带来显著优势。

4.3.2 计算开销。图9(a,b)展示了 CatapultDB 和基线方法在查找期间于邻近图中访问的平均节点数，并对比了不同束宽 k 以及 $t = 4$ 和 $t = 16$下的表现。在所有设置下，CatapultDB 都减少了平均访问节点数，表明弹射器能够有效降低查询延迟，但这一降幅不如图 6(a,b) 中明显。对于 $t = 4$ 和 $k = 1$，CatapultDB 的平均访问节点数比表现最佳的基线方法少 35.7%；对于 $k = 16$，这一降幅为 8.1%。图9(c)展示了所评估系统在查找期间计算的平均距离，并对比了不同束宽 k下的表现。该图表明，CatapultDB 所需的距离计算次数少于基线方法，而且随着 k 增加，这种降幅略有扩大。

4.3.3 弹射器使用情况。图9(d)展示了使用弹射器的查询数量随 k 变化的情况。对于 $k = 1$，CatapultDB 的使用率为 98.1%；对于 $k = 16$，则降至 97.6%。值得注意的是，与偏置工作负载相比，使用无偏查询工作负载时，CatapultDB 的命中率更高（见图 6(d)）。我们认为，这是因为均匀分布的查询会均匀散布到各个 LSH 桶中，因此每个桶都会积累弹射器目标节点，并且这些目标节点可以被频繁重复使用。相比之下，偏置工作负载会将流量集中在少数热点桶中，而许多其他桶中的查询则较为稀疏，这可能会降低能够找到相关弹射器的查询总体比例。尽管如此，我们注意到，在所有实验和工作负载中，弹射器的使用率始终高于 90%，因此得出结论：弹射器的使用频率非常高。

4.4 CatapultDB 与 DiskANN 在过滤查询下的性能

现在，我们使用 Papers 工作负载，评估 CatapultDB 和 DiskANN 在存在过滤查询时的性能。图10展示了两个系统在不同束宽下的吞吐量（每秒查询数）、平均召回率以及每次查询的平均距离计算次数。图10(a)表明，CatapultDB 的吞吐量高于 DiskANN： $k = 1$时高出 38.47%。随着 k增大，这一吞吐量相对优势会缩小；例如， $k = 16$时吞吐量提升了 22.10%。这一趋势在我们的实验中始终如一。与此同时，图10显示，在不同 k取值下，CatapultDB 的召回率都略高于 DiskANN， $k = 1$时最高高出 11.01%。图10(c)表明，在所有 k取值下，CatapultDB 所需的距离计算次数都少于 DiskANN。这一减少反映了弹射器带来的更短遍历路径：通过将搜索起点设在更靠近相关过滤后邻域的位置，CatapultDB 减少了跳数，从而减少了收敛前所需的距离计算次数。图10中的结果证实，即使在过滤搜索中，弹射器也常常能够发挥作用：过滤谓词只会禁用部分弹射器目标节点，其余有效目标节点足以为大多数查询提供显著的起点优势。

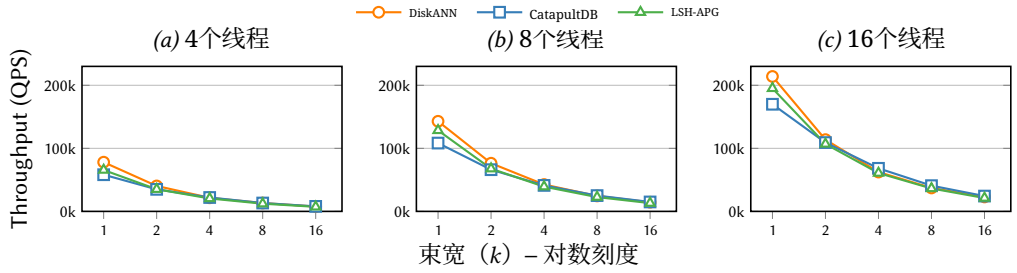

图 8：改变束宽 k 和线程数时，CatapultDB 与基线方法的吞吐量（每秒查询数）。我们使用均匀查询工作负载。

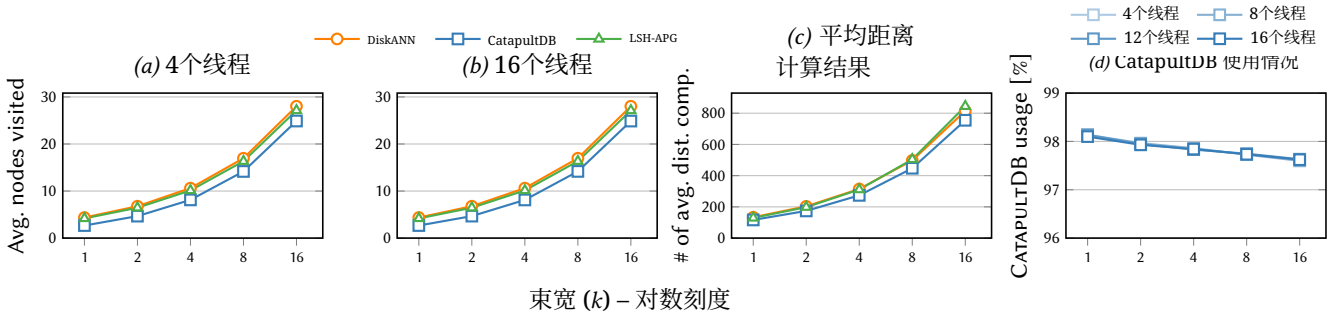

图9：(a,b) 改变束宽和线程数时，CatapultDB 与基线方法在邻近图中访问的平均节点数。(c) 改变束宽 k时，CatapultDB 与基线方法计算的平均距离数。(d) 改变束宽和线程数时，查找期间使用弹射器的查询比例。我们使用均匀工作负载。

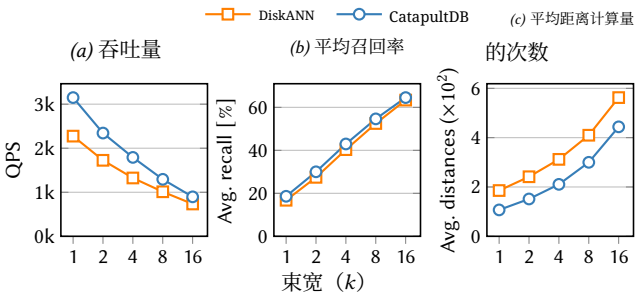

图10：改变束宽时，CatapultDB 与 DiskANN 的吞吐量（每秒查询数）、平均召回率和平均距离计算次数。

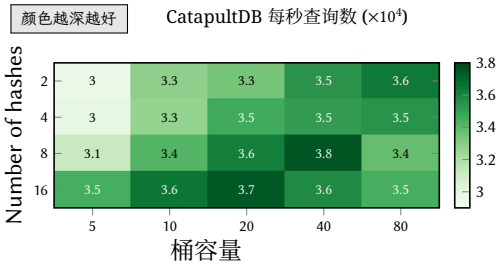

图11：桶容量 b 和LSH超平面数量 L 对CatapultDB吞吐量的影响，使用TripClick工作负载。

4.5 CatapultDB超参数对吞吐量的影响

我们评估 CatapultDB 对两个超参数的敏感性：LSH 超平面的数量 L 和桶容量b。在此实验中，我们使用 TripClick 工作负载，设置 $t = 8$ 个线程和 $k = 8$的束宽。图11以热力图展示了所评估的所有（b，L）组合的平均吞吐量（每秒查询数）。在所有评估配置中，$b = 40$ 和 $L = 8$ 始终能够提供最高或接近最高的吞吐量，平均为 37552 QPS

在三个随机种子下，吞吐量为 29701 每秒查询数，相比性能较弱的配置（$b = 5, L = 2$，29701 每秒查询数）提升了 26%。因此，我们将这些值选作本文所有其他实验的默认配置。最优点附近的性能表现稳定：$b \in \{20, 40\}$ 和 $L \in \{8, 16\}$ 的配置都能达到接近峰值的吞吐量，这表明 CatapultDB 无需精确的超参数调优，也能实现出色的性能。

5 相关工作

基于图的近似最近邻索引。基于图的方法是近似最近邻搜索的主流范式 [19, 34]。这类方法在数据集上构建邻近图，并从入口点开始通过贪心遍历回答查询。其核心挑战是构建一个既稀疏（每个节点的度较低，从而限制每跳的工作量）又易于导航的图（贪心搜索能够迅速收敛到最近邻）。

NSW [22] 通过逐个插入向量并将每个新向量连接到当前图中的近似邻居，逐步构建图。这种方法速度很快，但可能产生度极高的枢纽节点，从而拖慢遍历。HNSW [23]通过将图组织成多个层级，并为每一层的度设置上限来解决这一问题，避免任何单个节点成为瓶颈；它可以说是生产环境向量数据库中部署最广泛的索引 [25]。NSG [8] 和NSSG [7] 致力于构建更稀疏且可导航性保证更强的图，而HVS [21] 和HCNNG [32] 则分别通过基于Voronoi和基于聚类的策略降低构建成本。

Vamana [14] 引入了一个剪枝参数 $\alpha > 1$，明确地以略微增加度为代价，大幅缩小图直径，从而减少每次查询所需的跳数。这是DiskANN [14], 的关键支撑：它将图存储在固态硬盘上，并仅在内存中保留压缩向量；跳数减少会直接减少磁盘读取，使DiskANN能够在商用硬件上扩展至数十亿个向量。FreshVamana [30] 扩展了DiskANN，支持并发插入和删除，同时在长期更新过程中保持召回率。FilteredVamana [11]进一步加入标签感知图构建和搜索，使其能够执行基于属性的过滤查询，同时不牺牲图遍历的效率。

在所有这些系统中，只有索引数据发生变化时才会创建或删除边，从不会根据查询工作负载进行调整。据我们所知，CatapultDB 是首个能够根据观察到的搜索轨迹动态重组边的系统；它与 FreshDiskANN、FilteredVamana 等动态和过滤变体互为补充。

混合*LSH*-图方法。LSH-APG [36] 通过轻量级 LSH索引增强近邻图。在搜索时，首先查询 LSH结构，找到一个已接近查询的入口点，从而缩短后续的图遍历。它还利用基于 LSH 的距离界限，在遍历过程中剪枝远距离候选项。然而，LSH-APG 不支持过滤搜索，尚未证明其能够扩展至十亿级规模，而且需要构建专用索引：它无法叠加在 DiskANN 这类现有图数据库之上。CatapultDB 无需辅助结构，其捷径源自在线工作负载，而非索引构建时的静态哈希。

学习得到的捷径。 SHG [12] 在线下训练分段线性模型，以预测针对给定查询可以安全跳过 HNSW 层级结构中的多少层，从而避免对中间层进行冗余遍历，并实现 1.5–1.8× 倍加速。与 CatapultDB 不同，SHG 专用于分层索引，并在构建时根据数据分布学习捷径，不依赖查询工作负载。这两种方法互为补充：SHG 可以缩短跨层级的垂直导航，而 CatapultDB 则通过捷径加速单个层级内的水平遍历。

基于缓存的策略。 Proximity [2] 在向量数据库之前放置一个近似缓存：如果新的查询嵌入与先前出现过的查询之间的距离处于距离阈值 τ 之内，则直接返回缓存的邻居列表，从而完全绕过数据库。这可以消除高度重复查询的搜索成本，但引入了一个必须针对每个工作负载进行调优的阈值，无法与过滤或动态插入组合使用，并且在查询分布发生变化时需要刷新。CatapultDB 则将捷径边注入图本身，从而在不使用外部缓存的情况下继承底层索引的全部功能。这两种策略互为补充：当查询几乎完全重复时，Proximity 表现更优，而 CatapultDB 则缩短图遍历。

6 结论

我们提出了 CatapultDB，一种轻量级机制，它根据观察到的搜索轨迹，动态地将捷径边（即弹射器）注入邻近图。与以往的基于图的索引结构不同，CatapultDB 率先在运行时重组其边集，以利用查询工作负载的空间局部性和时间局部性。CatapultDB 作为任何接受可配置入口点的索引之上的透明层运行，保留了底层系统的全部功能集，*e.g.*，过滤搜索、动态插入和磁盘驻留索引，而无需重建索引或对底层搜索算法做任何修改。我们在四种工作负载上的实验表明，作为 DiskANN 之上一层的 CatapultDB 在同等或更高召回率下可将吞吐量最高提升 2.51× 倍，并在有偏工作负载上将每次查询的平均访问节点数最多减少 66.3%。CatapultDB 的搜索效率也达到或超过 LSH-APG，同时还支持 LSH-APG 无法处理的过滤查询和分布外插入。在过滤搜索下，CatapultDB 的吞吐量比 DiskANN 最高高出 38.47%。在完全均匀、无偏查询工作负载的最坏情况下，CatapultDB 不会带来有意义的吞吐量退化。因此，CatapultDB 代表了迈向工作负载感知向量索引的一步，我们预计利用查询局部性可以成为下一代向量搜索系统中的一项标准设计考量。

致谢

本研究由瑞士国家科学基金会资助，项目名称为“FRIDAY：节俭、注重隐私且实用的去中心化学习”，SNSF资助编号为10001796。

参考文献

[1] Sonu Airen 和 Jitendra Agrawal。2022。使用 k 近邻变体的电影推荐系统。《国家科学院科学快报》45，1（2022），75–82。[2] Shai Bergman、Anne-Marie Kermarrec、Diana Petrescu、Rafael Pires、Mathis Randl、Martijn de Vos 和 Ji Zhang。2025。利用近似缓存加速检索增强生成（*Middleware '25*）。https://doi.org/10. 1145/3721462.3770776[3] Steve Chien 和 Nicole Immorlica。2005。利用时间相关性衡量搜索引擎查询之间的语义相似度。载于第*14*届万维网国际会议论文集。2–11。

[4] Haya Diwan、Jinrui Gou、Cameron Musco、Christopher Musco 和 Torsten Suel。2024。面向高维最近邻搜索的可导航图：构造与局限。《神经信息处理系统进展》37（2024），59513–59531。[5] Ophir Frieder、Ida Mele、Cristina Ioana Muntean、Franco Maria Nardini、Raffaele Perego 和 Nicola Tonellotto。2024。对话式搜索中的历史嵌入缓存。《ACM网络汇刊》18，4（2024），1–19。[6] Jerome H Friedman、Jon Louis Bentley 和 Raphael Ari Finkel。1977。一种以对数期望时间寻找最佳匹配的算法。《ACM数学软件汇刊》（TOMS）3，3（1977），209–226。[7] Cong Fu、Changxu Wang 和 Deng Cai。2022。基于卫星系统图的高维相似性搜索：效率、可扩展性与对未索引查询的兼容性。《IEEE模式分析与机器智能汇刊》44，8（2022）。https://doi. org/ 10 1109 2021 3067706 . /TPAMI. . Cong Fu、Chao Xiang、Changxu Wang[8]和 Deng Cai。2019。利用导航扩散图进行快速近似最近邻搜索。《VLDB会议录》 12, 5 2019 10 14778 3303753 3303754 ( )。https://doi.org/ . / . [9] Aristides Gionis、Piotr Indyk 和 Rajeev Motwani。1999. 通过哈希进行高维相似性搜索（VLDB ’99）。https://www.vldb.org/conf/1999/P49. pdf[10] Anıl Eren Göçer、Ioanna Tsakalidou、Hamish Nicholson、Kyoungmin Kim 和 Anastasia Ailamaki。2026。QVCache：查询感知向量缓存。arXiv预印本 arXiv:2602.02057（2026）。[11] Siddharth Gollapudi、Neel Karia、Varun Sivashankar、Ravishankar Krishnaswamy、Nikit Begwani、Swapnil Raz、Yiyong Lin、Yin Zhang、Neelam Mahapatro、Premkumar Srinivasan、Amit Singh 和 Harsha Vardhan Simhadri。2023。Filtered-DiskANN：带过滤条件的近似最近邻搜索图算法（WWW ’ *23* 10 1145 3543507 3583552 ）。https://doi.org/ . / . [12] Zengyang Gong、Yuxiang Zeng 和 Lei Chen。2025. 加速层次图中的近似最近邻搜索：利用捷径实现高效层级导航。《VLDB会议录》18，10（2025）。https://doi.org/10.14778/3748191. 3748212[13] Piotr Indyk 和 Rajeev Motwani。1998。近似最近邻：迈向消除维度灾难。载于第三十届 ACM 计算理论年度研讨会论文集。604–613。[14] Suhas Jayaram Subramanya、Fnu Devvrit、Harsha Vardhan Simhadri、Ravishankar Krishnawamy 和 Rohan Kadekodi。2019。DiskANN：在单个节点上进行快速、精准的十亿级点最近邻搜索。载于 NeurIPS。https://proceedings.neurips.cc/paper_files/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf[15] Qiao Jin、Won Kim、Qingyu Chen、Donald C Comeau、Lana Yeganova、W John Wilbur 和 Zhiyong Lu。2023。Medcpt：利用大规模 PubMed 搜索日志进行零样本生物医学信息检索的对比预训练Transformer。《生物信息学》39，11（2023），btad651。[16] Jeff Johnson、Matthijs Douze 和 Hervé Jégou。2019。利用 GPU 进行十亿级相似性搜索。《IEEE大数据汇刊》7，3（2019），535–547。[17] Herve Jégou、Matthijs Douze 和 Cordelia Schmid。2011。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（2011）。https://doi.org/10.1109/TPAMI.2010.57[18] Patrick Lewis、Ethan Perez、Aleksandra Piktus、Fabio Petroni、Vladimir Karpukhin、Naman Goyal、Heinrich Küttler、Mike Lewis、Wen-tau Yih、Tim Rocktä schel 等。2020。面向知识密集型自然语言处理任务的检索增强生成。《神经信息处理系统进展》33（2020），9459–9474。[19] Wen Li、Ying Zhang、Yifang Sun、Wei Wang、Mingjie Li、Wenjie Zhang 和 Xuemin Lin。2020。高维数据上的近似最近邻搜索——实验、分析与改进。《IEEE知识与数据工程汇刊》32，8（2020）。https://doi.org/10.1109/TKDE.2019. 2909204[20] Ting Liu、Charles Rosenberg 和 Henry A Rowley。2007。利用大规模最近邻搜索对数十亿张图像进行聚类。载于2007年IEEE计算机视觉应用研讨会（WACV’07）。IEEE，28–28。[21] Kejing Lu、Mineichi Kudo、Chuan Xiao 和 Yoshiharu Ishikawa。2021。HVS：基于 Voronoi 图、用于解决近似最近邻搜索问题的层次图结构。《VLDB会议录》15，2（2021）。https://doi.org/10. 14778/3489496.3489506[22] Yury Malkov、Alexander Ponomarenko、Andrey Logvinov 和 Vladimir Krylov。2014。基于可导航小世界图的近似最近邻算法。《信息系统》 45 2014 10 1016 2013 10 006 ( )。https://doi.org/ . /j.is. . . Yu A. Malkov[23]和 D. A. Yashunin。2020. 使用分层可导航小世界图进行高效、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）。https: //doi.org/10.1109/TPAMI.2018.2889473[24] Jason Mohoney、Anil Pacaci、Shihabur Rahman Chowdhury、Umar Farooq Minhas、Jeffery Pound、Cedric Renggli、Nima Reyhani、Ihab F Ilyas、Theodoros Rekatsinas 和 Shivaram Venkataraman。2024。流式向量搜索的增量式 IVF 索引维护。arXiv预印本 arXiv:2411.00970（2024）。[25] James Jie Pan、Jianguo Wang 和 Guoliang Li。2024。向量数据库管理系统综述。《VLDB期刊》33，5（2024）。https://doi.org/10.1007/ s00778-024-00864-x[26] Casper Petersen、Jakob Grue Simonsen 和 Christina Lioma。2016。信息检索中的幂律分布。*ACM 信息系统汇刊 (TOIS)* 34, 2 2016 1 37 10 1145 2816815 ( ), – 。https://doi.org/ 。/ [27] Pinecone Systems, Inc.。2026. Pinecone。https://www.pinecone.io。[28] Navid Rekabsaz、Oleg Lesota、Markus Schedl、Jon Brassey 和 Carsten Eickhoff。2021。TripClick：大型健康网络搜索引擎的日志文件*(SIGIR ’21)*。https://doi.org/ 10 1145 3404835 3463242 。/ 。Craig Silverstein、Hannes Marais、[29] Monika Henzinger 和 Michael Moricz。1999。超大型网络搜索引擎查询日志分析。载于*ACM SIGIR* 论坛，第 33 卷。美国纽约州纽约市 ACM，6–12。[30] Aditi Singh、Suhas Jayaram Subramanya、Ravishankar Krishnaswamy 和 Har- sha Vardhan Simhadri。2021。FreshDiskANN：用于流式相似性搜索的快速、准确的基于图的近似最近邻索引。arXiv: 2105 09613 。Bing Tian、[31] Haikun Liu、Zhuohui Duan、Xiaofei Liao、Hai Jin 和 Yu Zhang。2024。使用{SmartSSDs}实现可扩展的十亿级数据点近似最近邻搜索。载于*2024 USENIX* 年度技术会议 *(USENIX ATC 24)*。1135–1150。[32] Javier Vargas Muñoz、Marcos A. Gonçalves、Zanoni Dias 和 Ricardo da S. Torres。2019。用于大规模近似最近邻搜索的基于层次聚类的图。模式识别 96 (2019)。https://doi.org/10.1016/j.patcog. 2019.106970[33] Jianguo Wang、Xiaomeng Yi、Rentong Guo、Hai Jin、Peng Xu、Shengjun Li、Xi- angyu Wang、Xiangzhou Guo、Chengming Li、Xiaohai Xu 等。2021。Milvus：专为向量数据管理打造的系统。载于*2021* 年国际数据管理会议论文集。2614–2627。[34] Mengzhao Wang、Xiaoliang Xu、Qiang Yue 和 Yuxiang Wang。2021。基于图的近似最近邻搜索的全面综述与实验比较。*VLDB* 会议录 14, 11 (2021)。https: //doi.org/ 10 14778 3476249 3476255 。/ 。Weaviate B.V.。[35] Weaviate。https://weaviate.io。2026. [36] Xi Zhao、Yao Tian、Kai Huang、Bolong Zheng 和 Xiaofang Zhou。2023。迈向高维空间中的高效索引构建与近似最近邻搜索。*VLDB* 会议录 16, 8 (2023)。https://doi.org/10.14778/ 3594512.3594527