知识图谱中的高吞吐量向量相似性搜索

Jason Mohoney*†、Anil Pacaci*、Shihabur Rahman Chowdhury*、Ali Mousavi*、
Ihab F. Ilyas*、Umar Farooq Minhas*、Jeffrey Pound*、Theodoros Rekatsinas*
*苹果公司、†威斯康星大学麦迪逊分校

摘要

机器学习越来越多地用于将数据编码为向量，以支持在线推荐和搜索应用。因此，近期的数据管理系统提出在查询处理中加入在线向量相似性搜索。在这项工作中，我们探讨了知识图谱（KG）场景中的向量相似性搜索。受寻找相关知识图谱查询以及过往知识图谱查询工作负载中的实体等任务启发，我们重点研究混合向量相似性搜索（简称混合查询）：查询的一部分对应向量相似性搜索，另一部分则对应底层数据向量相关关系属性上的谓词。例如，给定针对某个歌曲实体的过往知识图谱查询，我们希望为新的歌曲实体构建新查询，使其向量表示与过往知识图谱查询中该实体的向量表示接近。但知识图谱中的实体也具有非向量属性，例如歌曲关联的艺术家、流派和发行日期。因此，推荐的实体除了满足基于向量的相似性谓词外，还必须满足针对非向量属性的查询谓词。尽管这些任务是知识图谱的核心任务，我们提出的方法也普遍适用于混合查询。不同于以往优化在线查询的研究，我们专注于实现对过往混合查询工作负载的高效批处理。我们提出了用于高吞吐量批处理混合查询的系统 HQI。我们引入一种工作负载感知的向量数据分区方案，以便根据给定工作负载定制向量索引布局，并介绍一种多查询优化技术，以降低向量相似度计算的开销。我们在工业工作负载上评估了这些方法，并证明，与现有混合查询处理方法相比，HQI 在查找相关知识图谱查询方面可使吞吐量提升 31× 。

CCS 概念

• **信息系统** → **数据管理**；信息检索查询处理。

ACM 引用格式：
Jason Mohoney、Anil Pacaci、Shihabur Rahman Chowdhury、AliMousavi、Ihab F. Ilyas、Umar Farooq Minhas、Jeffrey Pound 和 Theodoros Rekatsi-nas。《知识图谱中的高吞吐量向量相似性搜索》。载于*ACM* 数据管理国际会议论文集（*SIGMOD '23*）。ACM，纽约州纽约市，美国，共 13 页。https: //doi.org/XXXXXXX.XXXXXXX

允许免费制作本作品全部或部分内容的数字或纸质副本，供个人或课堂使用，前提是副本不得用于营利或商业用途，也不得为此进行复制或分发，并且副本首页须保留本声明及完整引文。对于本作品中由 ACM 以外的其他方拥有版权的部分，必须尊重其版权。注明出处后，允许摘录。若要以其他方式复制、重新出版、发布到服务器或分发至列表，须事先获得特定许可和/或支付费用。请向 permissions@acm.org 申请许可。*SIGMOD '23*，*2023*年*6*月*18*日至*23*日，美国华盛顿州西雅图© 计算机协会。ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00 https://doi.org/XXXXXXX.XXXXXXX

1 引言

向量相似性搜索已成为推荐系统和搜索引擎中的关键组成部分。现代机器学习模型越来越多地用于将图像、文本或知识图谱中的实体等复杂数据嵌入为能够保留语义信息的向量表示[5, 6, 9, 18, 29]。对这些向量进行相似性搜索，可实现更准确、更具上下文针对性的搜索 [10, 13, 15, 36] 和推荐 [26, 27, 31, 33, 41] ，覆盖多模态数据。由于向量嵌入具有普适性，近年来向量数据库 产品不断涌现。新一代数据管理系统（如 ADBV [44],Milvus [42], Pinecone [2], Vearch [25], Vespa [3] 和 Vectara [1],）通过向量相似性搜索原语增强查询处理，为需要搜索向量并找出与查询向量最相似向量的工作负载提供支持。

在本研究中，我们探讨面向工业级知识图谱（KG）的受约束向量相似性搜索，以支持各类应用。这些应用包括查找相关实体、执行链接预测，以及检测错误事实 [5, 19, 43]。这些工作负载需要对混合查询进行批处理，其中混合查询由两部分组成：（i）向量相似性搜索；（ii）对关系属性上的谓词进行求值。例如，考虑一种服务：输入一首歌曲，推荐相关艺术家。对于数百万首歌曲组成的集合，若要支持此类服务，我们可能希望预先计算出每首歌曲的所有相关实体——即在向量空间中相似且类型为艺术家的实体。当前方法侧重于在线查询处理，缺乏支持高吞吐量批处理所需的优化。为此，我们提出了一套系统优化方案，以实现面向工业级知识图谱的高吞吐量混合查询处理。虽然我们是在知识图谱相关任务的背景下介绍这些优化方案，但它们具有通用性，可纳入通用向量数据库系统中。

驱动工作负载与需求。 我们最近推出了 Saga [19], ，这是一个用于大规模构建和提供知识服务的平台。作为这项工作的一部分，我们使用知识图谱嵌入上的相似性搜索 来解决诸如为过去的用户查询查找相关知识图谱查询或实体 以及用于缺失事实补全的链接预测 等任务 [19]。

对于相关知识图谱查询，我们的目标是构建并预先评估一组与过去用户查询相关的知识图谱查询；系统处理的查询量为缓存提供了大量机会，因为不同用户会重复提交相同查询。这类服务能够带来更丰富的用户体验。例如，对于用户查询“泰勒·斯威夫特有多高？”，我们希望为与泰勒·斯威夫特相似或相关的人物构建“这个人有多高？”这类查询。对于链接预测，我们的目标是利用向量相似性搜索 [47]，通过推断新事实来丰富知识图谱例如，为“泰勒·斯威夫特”补充缺失的“合作者”事实。这些应用场景

具有以下一项或多项特征：

- **混合查询：**我们需要评估混合查询，它将向量相似度与关系型属性谓词结合起来。对于查询“泰勒·斯威夫特有多高？”，我们需要找出在知识图谱嵌入向量空间中与“泰勒·斯威夫特”接近的顶级实体，同时还必须满足以下谓词：实体类型等于“人物”，且“身高”属性的值不为空值。要对已识别的相关查询给出有效答复，最后两个谓词是必要的。同样，为“泰勒·斯威夫特”补充缺失的“合作者”事实，需要执行链接预测，在谓词“合作者”的条件下，识别类型为“艺术家”且接近“泰勒·斯威夫特”的实体。

- **批处理：**与在线相似性搜索不同，这些工作负载通常需要在批处理环境中处理，例如，为一批过去的用户查询查找相关查询，或对所有知识图谱实体执行链接预测 [19]。为确保高效处理，我们需要一种查询评估设计，使其像分析型工作负载一样优先考虑吞吐量。

- **先前工作负载的可用性：**我们发现，工业级问答知识图谱工作负载具有过滤器共性和过滤器稳定性，因此可以根据工作负载特征定制向量数据库设计 [38]。此外，混合查询中的关系谓词通常与用于计算相似度的向量相关（例如，属性“身高”与表示人物的实体相关）。因此，工作负载感知方法可以通过考虑这些属性来帮助优化向量搜索。

上述要求需要当前向量数据库系统尚未涵盖的新型优化。首先，大多数系统对关系型属性谓词的支持有限，即主要支持数值比较，而且未针对包含多个谓词的查询进行优化。其次，所有现有系统都将关系谓词与向量相似性搜索视为独立查询，再合并其结果以生成最终结果。正如我们将在第6节中展示的那样，这样做可能会造成不必要的计算开销或较低的召回率。第三，所有现有系统都旨在降低单个在线查询的延迟，而不优化吞吐量。然而，我们的许多知识图谱工作负载会批量评估混合查询，因此需要高吞吐量批处理。最后，历史查询工作负载的存在为利用数据和工作负载分布来设计针对特定用例的算法和数据结构提供了机会 [24, 38]。在关系型工作负载中，针对给定工作负载和数据定制系统组件，已证明能够显著提升性能，优于传统方案 [4, 7, 17, 38, 46]。据我们所知，没有向量数据库系统会利用先验工作负载信息进行工作负载感知的数据库设计。

在此，我们介绍所提出的混合查询索引（HQI）框架的关键组成部分，该框架旨在实现面向工作负载的混合查询高吞吐量批处理。我们的贡献聚焦于两个维度：（i）向量索引设计中的工作负载感知（工作负载感知与工作负载无感知）；以及（ii）查询处理设置（批处理与在线）。针对这两个维度采用优化措施所构成的空间如图1所示。在同一图中，我们还将能够从相应象限中的优化措施获益的知识图谱任务标示出来。现有向量数据库采用的索引设计局限于一个象限：向量索引不感知工作负载，且查询以在线方式处理。相比之下，我们的知识图谱相关工作负载使我们能够探索其余空间。例如，相关查询或实体服务可以利用工作负载感知搜索（例如，基于先验工作负载的在线相关知识图谱查询），以最大限度地缩短查询执行时间。同样，该服务和链接预测（在没有先验工作负载的情况下）都可以从批处理优化执行中获益。这两个优化机会相辅相成。为此，HQI 提出了一套解决方案，包括工作负载感知向量索引和多查询优化技术。这些解决方案覆盖了我们关注的优化空间。此外，我们还表明，与采用广泛使用的向量相似性搜索开源库 [23] 的基线方案相比，这些优化能够显著提升查询处理吞吐量。我们的贡献如下：

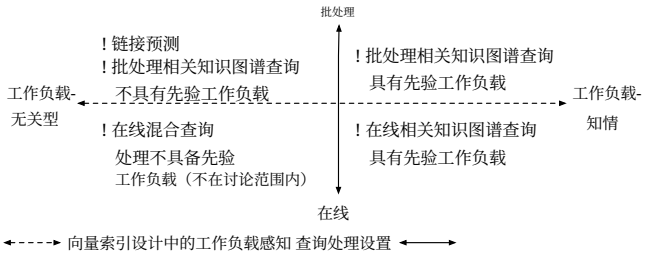


图1：向量搜索的优化空间及支持性知识图谱任务的分配。

工作负载感知向量索引（第 4 节）：首先，我们介绍一种面向向量数据库的新型工作负载感知索引。向量数据库通常使用专用向量索引，通过对数据进行分区或围绕质心构建多级索引，加速向量相似性搜索 [12, 42, 45]。在这里，我们利用过去的工作负载信息指导底层索引中的向量分区，使混合查询能够通过访问尽可能少的分区来完成。我们的方法受到以下事实的启发：目标混合查询工作负载中的关系谓词表现出过滤器共性和过滤器稳定性 [38],这两种特性，从而能够针对特定工作负载的特征优化数据布局。为此，我们扩展查询数据路由树（*qd*树）[46] 的概念，在数据加载时生成物理数据布局的过程中，同时考虑混合查询工作负载中的向量和关系谓词。此外，我们讨论了这种数据布局如何支持在查询运行时联合评估属性谓词和相似性谓词。作为独立优化方案，我们提出的工作负载感知向量索引方案显著减少了回答混合查询时扫描的元组数量；与工作负载无感知方法相比，在一个在线相关知识图谱查询用例中，吞吐量提升了 4× 。

批量查询优化（第 5 节）：其次，我们介绍一种多查询优化技术，该技术：(i) 对具有相似属性和向量相似度约束的查询进行批处理；(ii) 使用基于向量聚类的索引，对从中获取的向量倒排列表执行批量向量距离计算。这种优化的动机在于，我们的目标混合查询

表1：查询工作负载特征。行按选择率从低到高排序。

模板	历史快照（等长时间窗口）				可行知识图谱实体（%）
	t_0	t_1	t_2	t_3	
T1	15%	17%	17%	18%	< 0.005%
T2	26%	26%	26%	26%	<0.1%
T3	<1%	<1%	<1%	<1%	<0.1%
T4	24%	20%	20%	20%	<0.5%
T5	11%	12%	11%	12%	<0.5%
T6	2%	2%	2%	2%	<1%
T7	3%	3%	4%	3%	2.5%
T8	15%	15%	15%	14%	30%
T9	<1%	<1%	<1%	<1%	58%
T10	4%	4%	4%	4%	60%

工作负载需要在批处理环境中进行评估，从而实现跨查询的计算共享。这种优化适用于任何基于聚类的向量索引。与使用针对在线搜索优化的系统逐个执行查询相比，对于批量相关知识图谱查询（不考虑先前工作负载）和链接预测，我们的方法分别使批量查询处理吞吐量提升 17× 和 19× 。此外，与以往方法相比，将查询批处理技术与工作负载感知索引布局相结合，可使相关知识图谱查询用例的吞吐量提升至多 31× 。

最后，我们使用工业知识图谱工作负载中的混合查询和合成基准测试进行了若干微基准实验，并在讨论研究发现及面向向量数据库系统的新优化建议后结束全文（第6节）。

2 预备知识

我们将讨论本文后续部分所需的预备概念。首先，我们介绍推动本研究的工业知识图谱工作负载需求，并着重说明其关键特征（第2.1节）。随后，我们回顾处理混合查询的现有解决方案（第2.2节），并讨论这些方案在我们所关注的工作负载方面的局限性（第2.3节）。

2.1 工作负载特征

在第1节中，我们讨论了混合向量搜索如何在Saga中用于支持多种应用，例如查找相关实体、链接预测和检测错误事实 [19]。我们以相关知识图谱查询这一用例为贯穿示例，介绍必要概念。尽管如此，所有用例都具有与该用例相同或部分相同的特征，因此都能从本文提出的优化中受益。

相关知识图谱查询的应用场景需要分析知识图谱上的历史查询，以便为过去的查询推荐新的相关查询，或识别与输入的知识图谱查询相关的实体（参见第 1 节中的动机示例）。正如我们此前关于 Saga [19], 的工作所述，可以使用将知识图谱嵌入向量空间的技术即，为每个实体关联一个向量表示。我们还将知识图谱中两个实体之间的相关性定义为其向量表示之间的相似度。基于以上内容，这两项任务都对应于在知识图谱上执行一组混合查询即，查找满足关系约束合取的知识图谱实体，且这些实体的向量表示与作为输入提供给历史知识图谱查询的实体所关联的向量相似。

示例。沿用第 1 节中的示例，用户查询Q“泰勒·斯威夫特有多高？”需要一个针对实体“泰勒·斯威夫特”的“身高”谓词模板查询。要查找与该用户查询相关的查询，需要执行混合搜索，以识别向量与“泰勒·斯威夫特”的向量相似、具有非空值“身高”且类型为“人物”的实体。

为了支持相关知识图谱查询服务，我们利用过去的知识图谱查询构建混合查询工作负载。这些过去的知识图谱查询决定了谓词模板的相应分布，以及满足这些模板所对应关系谓词的知识图谱实体分布。对于构建的混合查询工作负载，我们使用术语选择率 来表示实体满足某个谓词的概率，即，选择率越低，满足该谓词的实体数量就越少 [37]。接下来，我们分析四个随机抽样并聚合的工业查询工作负载历史快照，每个快照均涵盖相等的时间窗口。抽样过程确保不改变每个快照内谓词模板的分布。表1总结了这些历史快照中排名前10的谓词模板的分布特征。我们的主要观察结果如下：

- 谓词模板的分布呈现出过滤器共性 [38], 即，少量谓词被大多数查询广泛使用。例如，工作负载中 ≈80%的查询仅对应四个模板。
- 谓词模板会随着时间反复使用，其分布表现出过滤器稳定性 [38], 即*</style}*，即在给定时间的查询中，大多数谓词模板过去已经出现过。此外，查询工作负载的组成在不同快照之间不会发生剧烈变化，即，表1中某一特定模板所占的比例在不同快照之间不会变化。
- 混合查询展现出多种谓词模板，这些模板包含超出简单数值比较的复杂属性约束。例如，在该工作负载中，一些常见的谓词模板包括对知识图谱实体属性进行集合成员检查和非 NULL 检查。
- 谓词模板展现出广泛的过滤能力。例如，满足选择率最低的谓词模板（T1）的知识图谱实体少于 0.005%。与此同时，满足选择率最高的谓词模板（T10）的知识图谱实体超过一半。
- 实体拥有的属性集合受其类型影响，因此不同实体类型之间的属性出现频率分布和相关性各不相同。

这些特征要求采用工作负载感知技术，以确保随着时间推移，在处理不同工作负载时始终保持高吞吐量。尤其是，得益于过滤器共性和过滤器稳定性，我们可以构建一次工作负载感知索引，并在之后重复使用。

2.2 混合查询的评估方法

一类新兴的数据管理系统提供查询处理原语，以支持混合查询 [1–3, 25, 42, 44]。给定一个混合查询，这些系统能够在向量数据库中搜索对象，其中（i）对象的向量表示与查询向量相似，且（ii）对象属性满足

结构化属性约束。现有系统通常将这两个子任务分开处理，并采用以下一种或多种评估策略：

策略 A – 穷举搜索：此方法在属性上构建传统关系型索引（例如，B 树索引），并使用索引扫描评估属性约束。随后，它穷举计算查询向量与所有满足属性约束的向量之间的距离，以执行相似性搜索。此策略在向量相似度和属性约束方面都能获得精确结果。

策略B – 先进行属性过滤，再进行向量搜索：此策略依赖数据库中的两个互不相交的索引：(i) 向量上的近似最近邻索引（ANN）[20, 21, 23, 28]；以及 (ii) 向量属性上的关系索引。与策略 A 类似，此方法首先评估属性约束以获得候选集。随后，它使用近似最近邻索引对所得向量执行相似性搜索。此策略通常通过根据候选向量的标识符（ID）生成位图来实现，以便在索引遍历期间过滤掉不满足属性约束的向量。

策略C——基于属性的分区：该方法根据一个频繁搜索的属性对向量进行范围分区，并在每个分区内构建一个近似最近邻索引，从而生成两级向量索引。对于包含分区属性谓词的混合查询，该方法首先确定其分配范围满足查询约束的分区，然后根据各符合条件分区的大小，在其中执行穷举搜索或近似最近邻搜索。

策略D——带后过滤的向量搜索：该策略在向量上构建近似最近邻索引，首先使用该索引执行向量相似性搜索，然后过滤掉不满足属性约束的候选项。

策略 A 是一种穷举式解决方案，可针对向量相似性搜索约束得到精确结果。策略 A 的性能取决于属性约束的选择率，因此，该方法适用于小规模数据集，或属性约束选择率很高的混合查询。策略 B、C 和 D 旨在通过为向量相似性搜索扩展专用索引，解决大规模数据集上的可扩展性限制，并且常用于工业界的混合查询处理解决方案 [1–3, 25, 42, 44]。上述所有方法都是为在线查询评估而设计的；正如我们接下来将讨论的那样，对于第 2.1 节所述特征的混合查询工作负载，它们无法满足高吞吐量批量评估的要求。

2.3 现有方法的局限性

未针对批处理进行优化：现有的混合查询处理方法针对在线查询执行进行了优化。尽管这些系统在功能上可以通过逐一处理查询来支持批处理工作负载，但无法利用批处理环境中可实现的数据布局和查询处理优化。此外，我们所针对的混合查询工作负载中，查询向量彼此之间以及属性上的关系谓词彼此之间都存在共性，因此可以将相似查询一起执行，以实现高效索引遍历，而在线查询处理解决方案并未考虑这一点。在第6节中，我们通过实证表明，与使用在线解决方案相比，仅批量查询优化就能使相关知识图谱查询工作负载的执行时间最多提升 17× 。

子查询的分离执行：现有系统通常将属性过滤和向量搜索视为两个独立的子查询，并合并其结果以生成最终结果。它们要么分别通过相应的属性索引和近似最近邻索引执行属性过滤和向量搜索，再合并结果（策略A、B和C）；要么在近似最近邻搜索后，将属性过滤作为后处理步骤（策略D）。第一种方法会产生不必要的计算开销，因为两个子查询的剪枝能力无法结合；对于原本会被属性过滤器剪枝的元组，向量搜索仍会进行冗余的向量相似度计算。后一种方法的召回率较低，因为近似最近邻搜索无法感知向量属性，而作为后处理步骤的属性过滤可能会剪除大量近似最近邻搜索结果。联合考虑向量搜索和属性过滤，可以在不牺牲结果质量的前提下消除冗余计算，从而实现高效的混合查询评估。

对属性的支持有限：现有策略对属性约束的支持有限，例如，属性约束仅限于数值比较或精确文本匹配 [25, 42, 44]。处理属性结构化约束的一种常见方法，是根据经常使用的数值属性对向量进行范围分区（策略C）。尽管当查询可路由至单个分区时，这种方法十分有效，但涉及非分区属性约束的查询必须在所有分区上进行评估。支持多个属性的存储开销会随属性数量线性增长，因为这需要针对每个属性复制并分区向量。

缺乏工作负载感知：现有解决方案针对在线查询执行进行了优化，却没有利用可用的工作负载信息。在现实世界的混合查询工作负载中，属性约束中的结构化谓词与表示现实世界实体的向量往往存在相关性。例如，实体类型为“Song”的歌曲“Billie Jean”的向量表示，很可能与其他歌曲的向量相似，而非与城市的向量相似。考虑这些相关性后，便可据此改进索引设计，从而显著提升批量查询执行的性能，正如我们在第6节所示。

3 混合查询的批处理

3.1 问题定义

定义 1（向量数据库）。 V表示由以下形式的元组（向量）构成的集合： $t = (id, e, a)$其中， $t.id$是元组标识符， $t.e$表示一个 d维实值向量， $t.a$表示与该元组关联的属性值，详见下文。

A 表示有限属性集，其中每个属性 A_i 的定义域为 D_i。一个元组可以关联 A中的一部分属性；不失一般性，我们将元组 t 的属性 $t.a$表示为一个 $|A|$维向量，其中 $t.a[i] \in D_i$ 表示元组 t 中属性A_i 的值（不存在的属性对应的值设为空值）。

我们既不对向量及其属性的生成方式作任何假设，也不对其尺寸作任何假设。设计一种

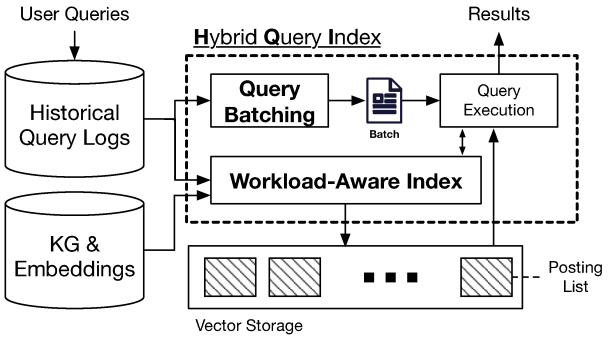


Figure 2: An overview of HQI for batch processing of hybrid vector similarity search queries.

embedding method with properly tuned parameters for a task such as related entities search is an orthogonal topic and is outside the scope of this work; a host of different models can be used [11, 14, 16, 32, 35, 39]. We also assume V to remain static over the course of batch query processing. Updates to V are accumulated over a longer time period and all indices used for batch query processing are rebuilt to reflect the updates.

Definition 2 (Hybrid Query (HVQ) Workload). *Q represents a workload of hybrid queries where each hybrid query $q \in Q$ contains the following: (i) a feature vector e, and (ii) an attribute constraint f as a conjunctive Boolean predicate, i.e., $f : p_1 \wedge \cdots \wedge p_k$. W.l.o.g., we assume that a predicate $p \in f$ is either a unary comparison ($A_i \otimes x$), a set membership check (A_i `IN` $\{x_1, \cdots, x_j\}$) or an existence check (A_i `IS NOT NULL`) where $\otimes \in \{<, \leq, >, \geq, =\}$ is a binary predicate, $A_i \in A$ is an attribute, and $x_j \in D_i$ is a value from A_i's domain.*

Definition 3 (Batch HVQ Workload Processing). *Given a vector database V and a hybrid query workload Q, the task of batch processing of a hybrid query workload can be modeled as:*

```
SELECT Q.e, Q.f, V.id as candidate FROM Q, V
WHERE IsFilterValid(Q.f, V.a)
ORDER BY Related(Q.e, V.e) LIMIT K
```

`IsFilterValid` *checks if the attributes of a tuple satisfy the constraints, i.e., $\bigwedge_{p_i \in Q.f} \big(p_i(V.a)\big)$ evaluates to* `True`*, and* `Related` *computes the vector similarity between the query vector $Q.e$ and a database vector $V.e$.*

We can use the above formulation to express the example from Section 2.1, *i.e.*, finding top-K related entities to "Taylor Swift" which also satisfy the "Height" predicate template, as follows:

```
SELECT V.id as candidate FROM V
WHERE 'Person' IN V.a['type'] AND V.a['height'] IS NOT NULL
ORDER BY Related(Q.e, V.e) LIMIT K
```

Here, $Q.e$ is the embedding vector for the KG entity corresponding to "Taylor Swift".

3.2 Solution Overview

Based on the observation that scanning tuples and computing similarities between their vectors is the main bottleneck for hybrid query execution, we aim to eliminate unnecessary tuple scans and vector similarity computations by limiting the search space. For this purpose, we introduce HQI, a hybrid query processing system consisting of: (i) a workload-aware index layout that utilizes the knowledge of past queries for eliminating unnecessary tuple scans; and (ii) a batching technique that maximizes computation sharing across a batch of queries. A recent trend in database systems is to custom fit system components to a given workload and data, in this way providing significant performance improvements over their traditional counterparts in the context of relational workloads [4, 7, 17, 38, 46]. We build on these advances and introduce workload-aware optimizations within HQI, which can benefit use cases exhibiting stable distribution of predicate templates over time such as the related KG query search use case.

HQI takes a vector database and a sample of a historical workload of hybrid queries as input and builds a vector index with a custom layout that is optimized to perform well for the given workload. HQI utilizes the distribution of vectors and relational predicates in the query workload to inform the index layout, and executes queries in batches based on the commonality of vector similarity and attribute constraints. HQI's workload-aware vector indexing scheme can be employed standalone to benefit hybrid query processing even in the online setting by reducing unnecessary tuple scans. Similarly, the standalone application of the query batching technique in HQI can benefit applications that require high-throughput evaluation of hybrid queries in the batch setting by pruning redundant vector similarity computations, regardless of the availability of *a priori* knowledge about queries. Figure 2 shows a birds-eye view of our solution. Two key components of our solution are:

Workload-aware vector index: We introduce a vector partitioning scheme that uses the distribution of the attributes associated with vectors, the attribute constraints, and similarity of vectors present in the hybrid query workload. Our proposed scheme produces a concise description of all tuples within each resulting partition, called *semantic description*, that helps decide if the tuples within that partition can answer a given hybrid query. We then use the resulting partitioning scheme to generate an index layout that enables us to process a batch workload of hybrid queries by accessing vectors from as few partitions as possible. For workloads where templates are stable over time, *i.e.*, only a fraction of templates are newly introduced over time, such as the industrial workload presented in Section 2.1, the resulting index can benefit unseen queries without the need for re-indexing.

Batch query execution: When executing a batch of hybrid queries using these partitions, we first batch together all queries with the same attribute constraint (*e.g.*, predicate templates in the related KG queries workload) and route those queries to the appropriate partitions based on the partitions' semantic descriptions. After routing queries to the relevant partitions, we perform a batch vector distance computation against the vectors within that partition using a single matrix multiplication. We then merge results from across partitions.

4 WORKLOAD-AWARE VECTOR INDEX

Existing indexes for vector similarity search commonly rely on partitioning to scale search for input queries. A common strategy is to group the database vectors into clusters, each represented by a centroid, and use these clusters to partition the underlying vector database. At query time, only the partitions whose corresponding centroids are the closest to the query vector are scanned.

当数据库中的向量与关系型属性相关联时，现有的混合查询处理方案会将基于属性的分区与向量分区方法相结合（第2.2节所述的策略 C）。此类分区通常仅考虑一个属性 ——通常是一个常被查询的属性——并使用范围分区，首先将数据库划分为一组基于属性的分区。然后，在每个分区内构建一个单独的向量索引，例如倒排文件索引（*IVF*） [21] 或分层可导航小世界（*HNSW*） [28] 。在这种分区方案下，若混合查询包含针对分区属性的属性约束，则可仅考虑满足该谓词的分区来执行查询，从而剪去不满足谓词的分区，有效缩小向量相似性搜索的范围。对于在线查询符合特定关系型属性的应用场景，这种方法可能有效；但它无法满足我们在本文中所针对的工作负载需求。请注意，我们旨在处理带有针对多个属性的谓词约束的混合查询工作负载（第2.1节）。当查询针对非分区属性施加谓词时，上述分区方案无法利用属性约束的剪枝能力。

要设计一种索引，以便在存在多个属性的情况下处理具有高剪枝能力的混合查询，需要回答以下技术问题：

- 如何根据数据集和工作负载定制分区与数据布局?
- 如何避免数据库向量在多个属性的各个分区间重复存储，从而尽可能缩小索引规模?
- 如何以统一的方式结合向量相似度与涉及多个属性的复杂属性约束?

下一节将介绍一种能够解决这些问题的工作负载感知向量索引设计。

4.1 工作负载感知分区与索引

算法1： ConstructBalancedQDTree

输入： P = 分区及其向量； Q = 查询;

函数 ConstructBalancedQDTree(P, Q)
- $C \leftarrow$ExtractCutPredicates(Q)
- 根节点 ← 初始化_qdtree_节点()
- **如果** $|P|$ > 最小_大小 **则**
  - 拆分_谓词 ← ∅
  - 左侧_拆分_大小 ← 0
  - **当** 左侧_拆分_大小 $\leq |P|/2$ **执行**
    - 谓词 ← GetMinCostPredicate(P, Q)
    - $C \leftarrow C \setminus$ {谓词}
    - 拆分_谓词 ←拆分_谓词 ∪ {predicate}
    - (P_L, P_R) ← P.split(split_谓词)
    - 左侧_分割_大小 ← $|P_L|$
  - $(Q_{left}, Q_{right}) \leftarrow Q.split(P_L.B, P_R.B)$
  - root.left ← ConstructBalancedQDTree(P_L, Q_L)
  - root.right ← ConstructBalancedQDTree(P_R, Q_R)
- **返回** root

我们的负载感知向量索引设计旨在针对给定的混合查询工作负载，为向量数据库生成分区；其中，查询的属性约束包含针对多个属性的各种谓词。为构建此类索引，我们扩展了 *qd*-tree 数据结构 [46],；该结构是经典 kd-tree [8] 的泛化，用于关系型

算法2： GetMinCostPredicate

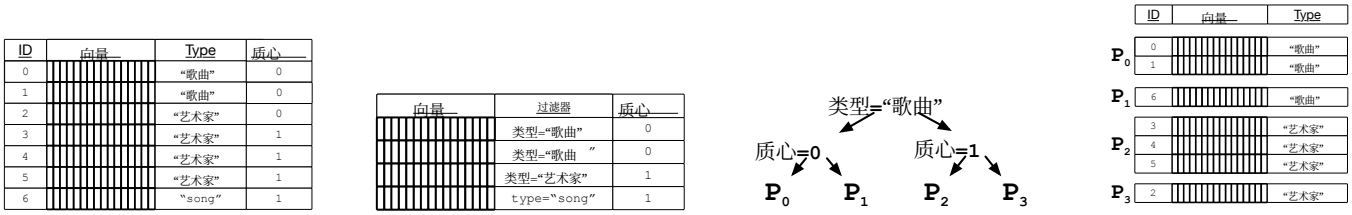
输入： P = 分区及其向量； Q = 查询;

函数GetMinCostPredicate(V， Q)
- $C \leftarrow$ExtractCutPredicates(Q)
- (最小_谓词，最小_成本) ← (nil， 2|Q|)
- **对于**谓词 $\in C$ **执行**
  - $(P_L, P_R) \leftarrow P.split(predicate)$
  - $(Q_L, Q_R) \leftarrow Q.split(P_L.B, P_R.B)$
  - $cost \leftarrow Q_L.size() + Q_R.size()$
  - **如果** $cost < min\ cost$ **那么**_
    - (min_谓词, min_cost) ← (谓词, cost)
- **返回** min_谓词

表的工作负载感知分区。简而言之，qd-tree 通过反复使用给定查询工作负载中的谓词切分元组，从而对表进行基于树的分区。qd-tree 的根节点对应整个数据集，并关联一个布尔切分谓词p；其左子树由满足 p 的元组组成，右子树则由满足 $\neg p$ 的元组组成。切分决策旨在尽量减少处理给定查询工作负载时需要访问的分区数量。给定一个 qd-tree，可从其叶节点生成数据集的不相交分区：对每个元组求值各节点上的谓词，并据此将元组从根节点路由至叶节点。每个分区中的元组都有一种简洁表示，称为 语义描述；它由该分区祖先节点的切分谓词生成，用于判断该分区中的元组能否回答给定查询。

qd树能够考虑给定工作负载中涉及多个属性的各种谓词，因此我们能够应对前两个挑战。将 qd树适配到混合查询工作负载的关键问题是，在做出切分决策时，同时考虑元组和查询上的向量相似度与属性约束。为此，我们设计了一种转换方法，将向量相似度约束表示为分类属性（第 4.1.1 节）。这样，我们就能以统一的方式表示向量相似度约束和属性上的关系谓词。随后，改进后的 qd树构建算法通过联合考虑跨多个属性的向量搜索谓词和结构化属性谓词，对向量进行分区（第 4.1.2 节）。这一方法的结果是一种分区索引设计，兼顾底层向量数据库以及给定混合查询工作负载中的谓词。最后，我们根据各分区的语义描述，将一个查询批次路由到相关分区进行处理（第 4.1.3 节）。

4.1.1 纳入向量相似度约束。qd树依赖从查询工作负载中提取的一元布尔谓词（范围谓词或相等谓词）来决定切分方式，并进行数据路由和查询路由。我们根据向量相似度约束，为查询向量和数据库元组添加一个类别属性，从而将向量相似度约束纳入 qd树的构建。首先，我们对V中的向量应用 k均值聚类，得到一组质心 C，并为每个质心分配一个整数标识符 $c.id$ 使得 $c.id \in [0, |C| - 1]$。然后，将 V 中的每个元组映射到相似度最高的质心，即向量空间中最近的质心，并将质心标识符用作表示该向量的类别属性。类似地，可以通过将向量表示为其最近的 m，把每个查询 $q \in Q$ 中的向量相似度约束转换为布尔谓词



(a) 向量与属性 **(b) 工作负载** **(c) 构建的 qd树** **（d）生成的分区**

图 3：（a）一个示例向量数据库；（b）混合查询工作负载；（c）基于工作负载（b）为（a）中的向量构建的示例 qd树；（d）基于（c）中的 qd树对（a）中的元组进行四路分区。

质心。直观而言，这会根据向量相似度为我们提供 V 中与给定查询相关的元组子集，其中参数 m 控制该子集的大小。转换后，每个元组 $t \in V$ 都有一个质心分配 $t.c$，每个查询则有一组 m 个质心分配 $q.c = \{c_0, c_1, \ldots c_m\}$。在本文其余部分中，我们分别用 $t.c$ 和 $q.c$ 表示元组 t 最近的质心，以及查询 q 最近的 m 个质心集合。图 3a 和图 3b 分别展示了 V 和 Q 的质心分配，其中 $|C| = 2$ 和 $m = 1$。

4.1.2 分区生成。给定一个混合查询工作负载，我们提取属性约束中出现的所有一元布尔谓词，并使用第 4.1.1 节所述的转换策略，从向量相似度约束中提取质心。例如，从图 3b 的示例工作负载中提取的切分谓词集合如下：

$$\{(t.A_{type} = \text{song}), (t.A_{type} = \texttt{artist}), (t.c \in \{c_0\}), (t.c \in \{c_1\})\}$$

qd树中的每个节点对应一个元组分区 V，并关联一个用于处理切分的语义描述 B。如 [46], 所示，qd树节点的语义描述是一个位图，用于标记该分区中的任意元组满足哪些已提取谓词。

最优 qd 树构建是一个计算上困难的问题 [46],，因此我们采用一种贪心构建流程，与 Yang 等人 [46]提出的方法类似。我们观察到，Yang 等人 [46] 提出的默认贪心策略会导致不平衡的 qd 树，且构建时间长、剪枝能力弱；也就是说，对于给定的混合查询工作负载，可跳过的分区数量较少。这是因为存在来自属性约束的高选择性谓词（如表1所示）以及质心分配。为此，我们提出了一种改进的贪心 qd 树构建算法（算法1中给出的 ConstructBalancedQDTree 过程），用于构建平衡的 qd 树。算法 1 从一个包含 V 中所有元组的分区开始，并递归地对节点进行平衡划分，直到达到最小分区大小 MIN_SIZE（第4行）。为确保划分平衡，我们的算法会从切分谓词集合中选择多个谓词，用于切分当前子树根节点，而不是只选一个。对于每个切分决策，算法1遍历谓词集合，并以贪心方式选出使切分成本最小的谓词（第7–12行）。回顾第3.1节，我们的目标是尽量减少用于评估 Q 的元组扫描次数。因此，贪心谓词选择旨在尽量减少评估混合查询时需要访问的分区数量

工作负载 Q（也就是说，最大化评估 Q 时可跳过的分区数量）。给定一个分区集合 $P = \{P_i, \cdots P_n\}$ 和一个工作负载 Q，$C(P, Q)$ 按如下方式计算需要访问的分区总数：

$$C(P, Q) = \sum_{P_i \in P} |P_i| \sum_{q \in Q} \begin{cases} 1, & Subsumes(P_i.B, q) \\ 0, & otherwise \end{cases} \quad (1)$$

这里，$|P_i|$ 对应一个分区中的元组数量 P_i，$P_i.B$ 对应一个分区的语义描述 P_i，Subsumes 是一个二元函数，用于指示在处理 q 时是否需要访问分区 P_i，即，P_i 的语义描述 $P_i.B$ 是否包含 q 的约束。算法1调用算法2中描述的 GetMinCostPredicate 过程，以最小化上述成本函数。对于工作负载 Q 中的每个谓词，算法2中的划分成本按需要路由到两个子树的查询数量计算，即，划分后需要访问两个分区的查询数量（第5–7行）。图3展示了一个基于示例数据库和工作负载构建的 qd树及数据库分区示例。

4.1.3 查询处理。给定算法1构建的 qd树，其叶节点定义了对原始数据库的一个互不相交且完整的分区。分区的语义描述用于确定评估给定查询时需要访问的分区集合。传入的查询会被路由到所有语义描述包含该查询的属性约束和质心分配的分区。请注意，这种基于 qd树的数据库分区与向量查询处理正交，并且可以在每个分区内创建独立的近似最近邻索引，类似于第2.2节所述的策略C。在本工作中，我们采用按分区构建的基于聚类的近似最近邻索引，即，在每个分区内创建独立的 IVF索引，并在语义描述包含查询谓词的各分区的 IVF索引上执行向量相似性搜索。

4.1.4 分析。 HQI构建包含两项任务：(i) 使用算法1构建一棵 qd树；以及 (ii) 为对应于qd树叶节点的每个分区训练一个 IVF索引。在最坏情况下，如果生成的qd树退化，算法1的复杂度将增长为 $O(|V|^2)$。只有当工作负载包含 $|V|$ 多个切分谓词，且每个谓词对 V 中的一组唯一的 $|V|-1$ 行求值为真时，才会发生这种情况。不过，这是一个病态示例；在实践中，我们没有观察到算法1生成的qd树出现此类不平衡。对于平衡qd树，算法1的复杂度将增长为 $O(|V| \log\ p)$，其中

p 是生成的qd树中的向量分区数量。对于每个这样的分区 P_i，训练一个包含 $|P_i|$ 个质心的IVF索引，时间复杂度为 $O(|P_i|\sqrt{|P_i|})$。对于平衡qd树，有 $\forall_i: |P_i| = \frac{|V|}{p}$。因此，构建 p 个IVF索引实例的总成本为 $O(|V|\sqrt{\frac{|V|}{p}})$。因此，为平衡qd树构建HQI的总体时间复杂度为 $O(|V|(\log p + \sqrt{\frac{|V|}{p}}))$。

4.2 感知属性约束的近似最近邻索引

第4.1节所述的工作负载感知索引策略通过剪枝与查询无关的分区，减少了扫描的元组总数。在每个分区内，我们采用基于聚类的近似最近邻索引来执行向量相似性搜索。通过为 V 中的元组属性构建结构化索引，并在两个索引之间使用侧向信息传递，我们可以消除近似最近邻索引中不必要的向量距离计算；即，将属性约束的精简表示推送到近似最近邻索引中。为此，我们在结构化索引（例如*B*树扫描）上评估属性约束，并构建一个位图来编码结果元组标识符集合。随后，我们将经过过滤的元组标识符的位图编码与查询向量一同传递给近似最近邻索引。我们修改底层近似最近邻索引中相应的搜索算法，使其利用此位图跳过那些因不满足属性约束而无论如何都会被丢弃的向量的距离计算。具体而言，我们修改基于聚类的索引（例如IVF）的索引扫描阶段，使其在扫描倒排列表时通过位图测试跳过向量距离计算。类似的策略也可应用于基于图的索引（例如HNSW）：在图遍历阶段使用位图测试，将向量排除在候选集之外 [12, 44]。

将位图纳入近似最近邻索引，以剪除不必要的向量距离计算，这与分区方案正交，也可以独立使用，将属性约束融入现有的向量相似性搜索技术。事实上，这与策略B（第2.2节）类似，并且在开源向量相似性搜索库和商业系统中，正逐渐成为在线评估混合查询的热门做法 [23, 42, 44]。独立应用这种方法的一个潜在局限在于，向量索引的构建不会考虑结构化属性。因此，评估带有高选择性约束的查询时，可能需要访问索引的大部分内容。与我们的工作负载感知索引布局相结合后，这种方法既能减少需要访问的索引部分，也能减少执行的距离计算次数，如第6节所示。

5 批量查询优化

第 4 节中的工作负载感知索引设计和查询处理策略利用了过去查询的信息，并通过跳过与给定查询无关的分区，减少元组扫描和向量距离计算的总次数。批量查询处理的特性为进一步消除冗余计算提供了优化机会，这种优化与工作负载索引相互独立。具体而言，我们从用于说明问题的工作负载中观察到，一部分查询向量和属性约束

算法 3： 批处理

```
输入： I = IVF索引； Q = 查询工作负载； k = 最近邻数量
       返回； nprobe = 每个查询要扫描的倒排列表数量
1 函数 批处理(I, Q, k, nprobe)
2     C ←提取谓词(Q)
3     R = 结果堆(num_结果=|Q|, max_大小=k)
4     对于过滤器 f ∈ C 执行
5         Q_f ← 分组(Q, f)
6         N ← 查找最近质心(Q_f, I, nprobe)
7         for 质心 c ∈ N do
8             Q_f^c = 分组(Q_f, c)
9             候选项 = 应用筛选 (I[c], 过滤器)
10            (邻居, 距离) = K近邻(Q_f^c, 候选项, k)
11            对于q ∈ Q_f^c执行
12                R[q.id].push(neighbors[q.id], dist[q.id])
13    返回 R.ids , R.values
```

被工作负载中的大多数查询所使用（第 2.1 节）。因此，对于给定的混合查询工作负载，我们可以根据查询所含向量和属性约束的相似性对查询进行批处理，并分批执行。通过在一批查询之间共享元组扫描和向量距离计算，我们可以在不影响结果质量的情况下提高吞吐量。

按照属性约束对查询进行批处理，是商业解决方案中常用的一种技术，可将针对一批具有相同属性约束的查询评估约束的成本分摊。鉴于少数模板及其对应属性约束频繁出现（如表1所示），这一策略对于我们的目标工作负载尤其有效。然而，基于向量相似度对查询进行批处理，则有机会通过共享近似最近邻索引遍历，消除冗余的元组扫描和向量距离计算；这些正是混合查询执行的主要瓶颈。考虑在每个 qd树 分区内创建的IVF索引。系统根据查询向量最近的质心，将其路由到一部分倒排列表，并对每个倒排列表进行穷尽扫描，以查找相似向量。我们的关键思路是，首先通过根据向量相似度对查询进行分组，找出需要在每个倒排列表上进行评估的查询子集，即依据它们在 IVF索引中的最近质心进行分组。然后，我们使用高效的矩阵乘法计算分组后的查询向量与倒排列表中向量之间的距离，而不是针对每个查询穷尽扫描倒排列表。

我们在算法 3 中概述了上述在 IVF索引上执行混合查询工作负载的批处理过程。给定一个混合查询工作负载 Q 以及一个针对分区P_i中的一组向量建立的 IVF索引 I，算法 3 按如下方式分批评估查询：

(1) 首先，根据查询属性约束中的谓词对 Q 中的查询进行分组（第 5 行）。(2) 在每个组内，为每个查询向量找到最近的 *nprobe* 个质心（第 6 行），并生成每个质心对应的查询向量列表（第 8 行）。(3) 对于每个倒排列表，获取满足属性约束的候选向量集合（第 9 行）。然后，通过一次矩阵乘法运算计算候选向量与查询向量之间的距离（第 10 行）。(4) 最后，使用矩阵乘法计算出的距离更新每个查询对应的最小堆，以跟踪每个查询的 k 个最近结果（第 12 行）。

6 实验结果

在评估中，我们使用工业界和公开可用的基准数据集，并将HQI 与近期的不同解决方案进行比较。我们首先介绍实验设置（第 6.1 节），然后展示 HQI 的端到端性能评估（第 6.2 节），对我们的技术贡献进行一系列微基准测试（第 6.3 节），并展示 HQI 对未见工作负载的泛化能力（第 6.4 节）。与现有解决方案相比，我们的成果亮点如下：

(1) 对向量进行工作负载感知分区与索引，可根据选择率将扫描的元组数量和向量距离计算次数最多减少 95%。(2) 所提出的查询批处理策略能够有效地在一批混合查询之间共享向量距离计算，使整体工作负载评估时间最多提升 19×。(3) 综合来看，HQI 可使整体工作负载执行时间最多提升 31×。

6.1 实验设置

表 2：评估数据集（n_q = 个查询向量）。

数据集	n_q	数据类型	指标	属性
SIFT-100M	10K	128 8位无符号整数	L2	合成
MSTuring-100M	100K	100 个 32位浮点数	L2	合成
YandexT2I-100M	100K	200 个 32位浮点数	IP	合成
LP	-	128 个 32 位浮点数	IP	实体类型
RelatedQS	-	128 个 32 位浮点数	IP	实体属性

数据集。 表 2 总结了公开数据集以及我们用于评估的随机采样工业数据集的属性。我们使用的公开数据集包括 SIFT-100M [34],MSTuring-100M 和 YandexT2I-100M；它们均来自 BIGANN 基准测试 [22]。这些数据集中的向量不包含任何属性，因此我们遵循先前的工作 [42, 44]，为向量生成合成属性和查询谓词。我们为数据集中的每个向量分配两个随机浮点属性 A 和 B。随后，我们生成 20 个范围谓词，其中 10 个针对属性 A，另外 10 个针对属性 B，使得给定列上的范围谓词的选择率 i 为 2^{-i}：$i \in [0, 9]$，即，2^{-i} 的向量满足该谓词 i。然后，我们通过对全部 20 个过滤器和数据集提供的所有 n_q 查询向量执行笛卡尔积来构建查询日志，共得到 $20 \cdot n_q$ 个查询。

对于工业工作负载，我们使用两个数据集：RelatedQS 和链接预测（LP）。对于 RelatedQS，我们使用部分知识图谱实体嵌入向量，并从匿名化的历史查询中随机采样并汇总得到查询工作负载。混合查询工作负载是通过将历史查询日志转换为（模板，实体）对获得的，其中模板是针对知识图谱实体属性的合取布尔谓词，实体是相应知识图谱实体的嵌入向量。对于 LP，我们通过从内部知识图谱中采样实体和关系来生成工作负载，以构成（模板，实体）对；不过，此处的模板对应的是针对实体类型的布尔谓词。这些工作负载所附带的查询集规模从数百万到数亿条查询不等。RelatedQS 代表一种包含历史查询的工业工作负载，因此可以同时受益于工作负载感知索引和批处理，而 LP 工作负载没有历史查询，因此只能受益于批处理。知识图谱实体的向量表示是通过在知识图谱上训练 GraphSage 模型 [14] 获得的。

我们在第6.2节的端到端评估中使用所有数据集，并在第6.3节使用 RelatedQS 和 MSTuring-100M 进行额外的微基准测试。对于使用 RelatedQS 的端到端评估，我们从其四个等长时间划分中选取一个划分（t_0 见表1）的所有查询来构建工作负载感知索引，并使用训练好的索引处理同一划分中的所有查询。所有实验均在单台 AWS r5.16xlarge 实例上运行，该实例配备 64 个虚拟CPU 和 512 GiB 动态随机存取存储器。

评估指标。 对于给定的混合查询工作负载，我们使用以下指标：1）总查询处理时间；2）处理该工作负载时扫描的元组数量；3）top-k 召回率。我们将召回率计算为真实值中包含的结果所占比例（通过穷举搜索获得，即第2.2节中的策略 A）。除非另有说明，我们报告 $Recall \geq 0.8$ 的运行时间，其中 $nprobe$表示扫描的倒排列表数量；该数量针对每个查询模板进行调整，以达到目标召回率。

对比方法。我们将提出的技术与第2.2节所述策略进行比较。具体而言，我们比较以下几种在多个商业解决方案中采用的混合查询处理策略 [1–3, 25, 42, 44]：

- **预过滤**：使用 FAISS [23] 开源库实现的 IVF 索引的策略B。
- **范围**：策略 C，其中向量依据一个常被搜索的属性进行范围分区，并在每个分区内分别创建基于 FAISS 的 IVF 索引。
- **后过滤**：使用 FAISS 实现的策略D，其中属性约束在向量搜索后进行评估。
- **HQI**：我们提出的工作负载感知方法，使用根据给定混合查询工作负载构建的 qd树对向量进行分区，并根据属性约束和向量相似度对查询进行批处理。

第2.2节中的策略 A 未予考虑，因为对所有向量进行穷举扫描对于我们实验中使用的大型数据集而言速度过慢。除非另有说明，所有基线方法默认均按属性约束对查询进行批处理，因为这是许多现有商业解决方案中常用的优化策略。同样，我们对所有方法都采用第4.2节所述的位图下推技术，以减少向量距离计算次数。FAISS 最近新增了将属性过滤器下推至近似最近邻索引以剪枝向量距离计算的功能[1]。我们使用此功能实现基于位图的 ID选择器，以便在需要时将属性过滤结果下推至 IVF 索引。此外，我们使用 $\sqrt{n}$ 个簇来训练 IVF 索引，其中 n 是给定分区中的向量数，或非分区索引对应的数据集中的向量数。对于 HQI，我们调整 m，并报告最佳配置下的运行时间（$m = 0$，其中不使用质心进行查询路由；如第6.5节所述，该配置在 RelatedQS 上具有最佳端到端性能）。

[1]https://github.com/facebookresearch/faiss/commit/dd814b5f146b9fb5f5f76070ff1e27b86e31a058[2]RelatedQS 和 LP 中的属性约束包含对多个属性进行存在性（IS NOT NULL）和成员资格（IN）检查，因此无法应用范围。

Table 3: Slowdown compared to HQI @ Recall >= .8

Approach	Dataset				
	RelatedQS	LP	MSTuring	SIFT100M	YandexT2I
HQI	1×	1×	1×	1×	1×
PreFilter	31×	19×	3.6×	0.97×	1.7×
PostFilter	136×	-	22×	4.1×	5.4×
Range	NA[2]	NA[2]	5.22×	1.2×	3×

Table 4: Index generation time compared to HQI

Approach	Dataset				
	RelatedQS	LP	MSTuring	SIFT100M	YandexT2I
HQI	1×	1×	1×	1×	1×
PreFilter	0.95×	1×	2.8×	2.15×	1.9×
Range	NA[2]	NA[2]	.85×	0.63×	0.58×

6.2 End-to-End Performance

We first demonstrate the performance benefits of HQI by comparing its end-to-end performance to existing approaches on all datasets listed in Table 2. Note that we augment each vector and query with synthetic attributes for MSTuring, SIFT100M, and Yandex T2I as described in Section 6.1.

Query Processing Time. Table 3 reports the relative workload execution time for all approaches with respect to HQI. We observe that HQI provides 31× and 19× speedup in workload execution time over the best performing baseline PreFilter on the RelatedQS and LP workloads, respectively. Note that Range cannot be used for RelatedQS and LP as their attribute constraints contain existence (`IS NOT NULL`) and set membership (`IN`) predicates over multiple attributes. We observe that PostFilter is significantly slower since its vector search stage does not utilize the pruning power of attribute constraints, requiring the vector search stage to return a large number of candidates to ensure the desired recall. For LP, we were unable to acheive the target recall using PostFilter.

We observe similar trends over the public datasets, where HQI matches or exceeds the best-performing baseline. In particular, we observe that PreFilter outperforms Range as only a portion of the queries, *i.e.*, queries with predicates over the partitioning attribute can utilize the additional pruning power of Range. The performance improvements of HQI over PreFilter and PostFilter are lower over these benchmarks compared to RelatedQS because of two reasons: (i) The synthetic predicates we generate do not have the variety and skewed distribution as RelatedQS (Section 2.1), limiting the effectiveness of workload-aware indexing. (ii) SIFT100M has 10× fewer queries and both YandexT2I and SIFT100M require scanning roughly 10× fewer posting lists per query to obtain a high recall, limiting the effectiveness of batching. A detailed analysis of how much each optimization in HQI contributes to its overall performance is presented in Section 6.3. Overall, HQI provides up to an order-of-magnitude improvement in workload execution time across industrial and public ANN benchmark workloads.

Runtime Breakdown per Query Template. Next, we turn our attention to the runtime breakdown for each query template from RelatedQS workload in Figure 4. Recall from Table 1 that the attribute constraints present in RelatedQS contain predicates over multiple attributes with a wide range of selectivities.

We find that across a majority of the templates, HQI consistently

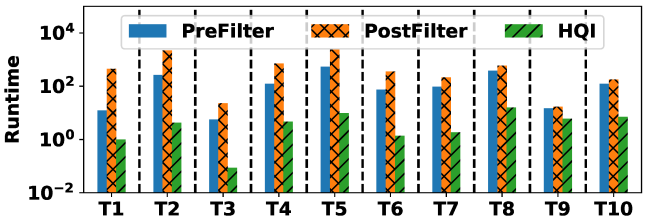


Figure 4: Runtime of RelatedQS Workload by query template. Runtime is normalized by the HQI runtime for template T1.

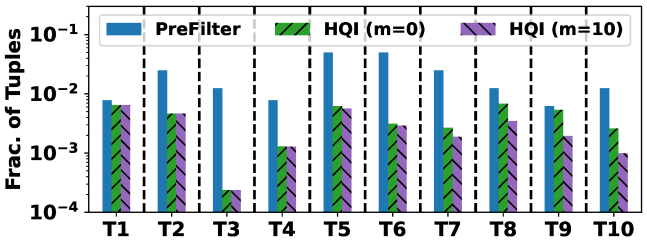


Figure 5: RelatedQS: Fraction of tuples scanned by template.

achieves more than 10× speedup in batch query execution time compared to the best baseline (PreFilter). Runtime differences are the most significant for low- to medium- selectivity templates (T1 - T8), whereas the improvement diminishes for high-selectivity templates (T9 - T10). This is because the number of candidate tuples that satisfy these high-selectivity templates is significantly larger than that of the other templates. Consequently, the effectiveness of partitioning for pruning tuples is limited. Albeit reduced, the runtime improvement remains substantial, with a 2.5× and 17× improvement for T9 and T10, respectively.

Index Generation Time. Table 4 shows the index generation time for each dataset normalized by that of HQI. We find that HQI's index generation time is similar to or faster than the baselines. While seemingly counter-intuitive, there are two reasons for this: (i) qd-tree construction accounts for 10% - 20% of index generation time and is not the dominant cost in index training; and (ii) training a partitioned IVF index scales better than training a single IVF index over the full dataset. The latter is due to the complexity of IVF index training: training a single IVF index over V scales as $O(|V|\sqrt{|V|})$ whereas HQI scales as $O\big(|V|(\log p + \sqrt{\frac{|V|}{p}})\big)$ as described in Section 4.1.4. The faster index generation time of HQI and Range indexes on the public datasets compared to PreFilter is a direct result of training multiple IVF indexes over partitioned vectors. For LP, there is no historical query log available for indexing so the qd-tree construction is skipped, therefore the HQI index generation is identical to PreFilter. Overall, qd-tree construction does not introduce significant overheads to index generation, and HQI can provide improved generation time over unpartitioned indexes.

6.3 Microbenchmarks

Here we present a detailed experimental evaluation of HQI by analyzing the performance gains of our workload-aware index design (Section 4) and query batching strategies (Section 5).

Reduction in Tuples Scanned: We first evaluate the pruning capability of HQI against the best performing baseline – PreFilter.

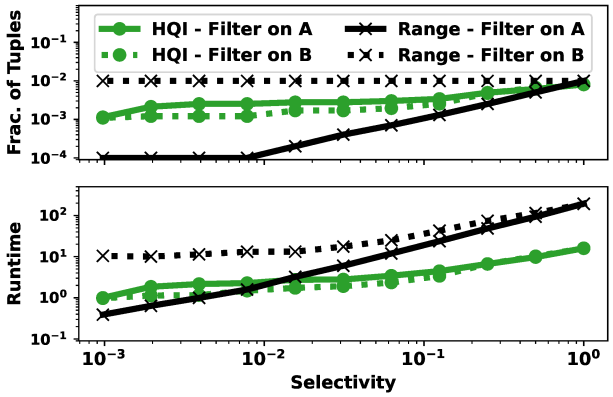


Figure 6: MSTuring: Normalized runtime and fraction of tuples scanned by selectivity.

We evaluate two configurations of HQI: $m = 0$ denotes that only attribute constraints are used, and $m = 10$ denotes that each query vector is mapped to the 10 closest centroids for partition generation decisions and query routing (Section 4.1.1). Figure 5 shows the fraction of tuples scanned to process a query for each template in the RelatedQS workload. Overall, we observe that a reduction in tuple scans corresponds to a reduction in runtime. HQI performs an order of magnitude fewer tuple scans than PreFilter for many templates, showing a 95% reduction for T3, in line with the runtime breakdown results reported in Figure 4. For templates with high selectivity (T9-T10), HQI ($m = 10$) provides additional pruning capability by skipping partitions that are unlikely to contain relevant vectors. However, using centroids for query routing introduces additional overhead due to finding the nearest centroids at query time, resulting in slower runtime for low- and medium- selectivity templates (T1-T7). This provides an interesting opportunity for an adaptive optimizer that controls the value of m based on query selectivity, which is further discussed in Section 6.5. Overall, for the RelatedQS workload, HQI with $m = 0$ and $m = 10$ results in 77% and 80% fewer tuple scans, respectively, compared to PreFilter.

Multi-attribute Partitioning: We now show HQI's effectiveness for hybrid query workloads containing predicates over multiple attributes. We compare HQI against the Range baseline over the MSTuring dataset where Column A is used for range partitioning and report the total number of tuple scans and batch query execution time in Figure 6. First, we observe that the Range baseline effectively reduces the tuple scans for queries with predicates over the partitioning attribute A. However, it cannot provide any pruning when the attribute constraint of a query does not contain any predicate over the partitioning attribute (*i.e.*, attribute constraint is a predicate over the non-partitioning attribute B), increasing runtime. In contrast, HQI consistently reduces the number of tuple scans, thus the execution time, regardless of the attribute present in the predicates. As opposed to Range, which partitions over a single attribute attribute, HQI considers all predicates present in the queries and partitions over multiple attributes, resulting in runtime improvements for queries with predicates over multiple attributes.

Query Batching: Here we analyze the effectiveness of batch query optimization strategies from Section 5. First, we evaluate the effect of batch size on HQI and the baseline PreFilter. We then implement combinations of attribute constraint- and vector similarity-based batching over a vanilla IVF index to isolate the impact of batching from the impact of workload-aware partitioning in HQI.

We first evaluate the impact of batch size on HQI compared to the baseline PreFilter, using template T4 in the RelatedQS workload. The runtime to process a single batch of query vectors to reach a target recall of 0.8 is shown in Figure 7a. The runtime is normalized by "Batch Size = 1" case for "HQI without vector batching" scenario. Our results show that HQI can process batches over 10× faster than PreFilter for query template T4 across all batch sizes. This is due to two reasons: (i) reduction in tuples scanned per query, and (ii) smaller filtering overhead. The first reason is illustrated in Figure 5, which shows that queries using template T4 require scanning roughly 10× fewer tuples due to the pruning capability of the qd-tree. The second reason is that filters and bitmaps need only be computed within the qd-tree partitions for HQI, rather than. across the full dataset as in PreFilter. Filtering overhead is especially detrimental for small batch sizes (*i.e.*, the online setting), where it can be the dominant cost. Our experiments also show that for small batch sizes, HQI without vector batching optimization is the best option, with a crossover point at a batch size of 100. This suggests that there is an opportunity for an optimizer to determine the best execution plan based on the batch size. Although our focus is on offline workloads, our results also show that HQI benefits the online setting (*i.e.*, Batch Size = 1). In summary, HQI can process queries up to 10× faster than PreFilter across batch sizes.

We next evaluate the impact of vector similarity-based batching using the MSTuring dataset without any attributes to demonstrate the batching technique's effectiveness for vector similarity search over a vanilla IVF index. Figure 7b presents the batch query execution time for achieving different recall levels by adjusting the number of posting lists (*nprobe*) scanned for each query. We observe that the runtime remains relatively stable with our batching strategy. This is in contrast to the IVF baseline, where achieving higher recall comes at the expense of a significant increase in query execution time (due to higher *nprobe*), *e.g.*, 21× slowdown at recall 0.9. With vector similarity-based batching, distances for a group of query vectors against a posting list are computed using a single matrix multiplication instead of pairwise distance computations.

Finally, we evaluate the effectiveness of attribute constraint-based batching across queries with various selectivities (Figure 7c). Here use augment vectors and queries in MSTuring-100M with synthetic attributes as described in 6.1. First, we observe that attribute constraint-based batching is a critical optimization for performant batch processing. It provides a 300× runtime improvement over issuing queries one at a time (No batching) by amortizing the cost of evaluating attribute constraints across a group of queries. Applying vector-similarity based batching (as in Algorithm 3) provides further 18× improvement, resulting in ≈ 5000× improvement over the naive strategy of issuing queries one at a time. In summary, our batching techniques are crucial to ensure high-throughput batch processing of hybrid queries without reducing recall.

6.4 Robustness to Future Queries

Here, we show that the workload-aware indexing in HQI can benefit unseen queries without the re-indexing for workloads where the

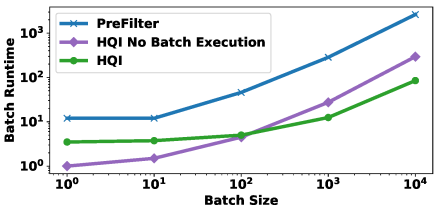


Figure 7(a): RelatedQS (Template T4): Normalized runtime to process a single batch with a varying number of query vectors in the batch.

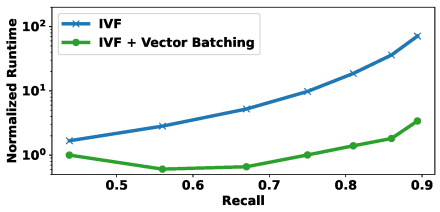


Figure 7(b): MSTuring (no attributes): Runtime vs. Recall using an IVF index with and without vector similarity-based batching.

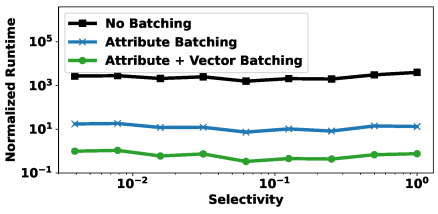


Figure 7(c): MSTuring: Runtime vs. Selectivity using an IVF index with and without attribute constraint-based batching.

distibution of predicates is relatively stable over time (workloads that exhibit *filter stability* as described in Section 2.1). For this purpose, we train HQI using predicates of all the queries from only one split of RelatedQS (t_0 from Table 1), and measure the workload execution time for each of the four equal temporal splits of RelatedQS ($t_0 - t_3$ in Table 1) using the trained index. We then compare it against the best performing baseline, PreFilter, which does not utilize past workload information during indexing.

We see from Table 5 that HQI can maintain over a 30× improvement in runtime over PreFilter across all splits. HQI can benefit unseen queries due to the relatively stable distribution of templates as shown in Table 1. As such, HQI trained over a representative query log can be re-used for future queries without the need for computationally expensive re-indexing. The small variation observed for t_0 through t_3 ($\leq$ 5% improvement compared to t_0) can be attributed to the differential improvement observed for the templates (Figure 4) combined with the variations in query workload composition (Table 1) in contrast to t_0.

Table 5: Queries per Second (QPS) for RelatedQS workload split into four equal time intervals, where HQI is trained over queries only from t_0. QPS is normalized by HQI at t_0.

Approach	Query Log Split			
	t_0	t_1	t_2	t_3
HQI	1×	1.05×	1.03×	1.05×
PreFilter	.032×	.031×	.032×	.032×

6.5 Discussion

Here we summarize the main findings of our evaluation and provide recommendations for new optimizations in vector database systems. First, we find that no single configuration of HQI provides the best performance across all queries and batch sizes. We observed in Figure 5 that using centroids during query routing (m = 10) is beneficial only to high-selectivity templates (T7-T10), while negatively affecting templates with lower selectivity due to the additional distance computations required. Furthermore, we observed in Figure 7a that there is a tradeoff point in query batch size where disabling vector-similarity based batching leads to reduced runtime for HQI. These observations show that there is an opportunity for an optimizer design that can choose the best execution policy based on the batch size and filter selectivity.

Through our experimental analysis, we find that our proposed query batching strategies are critical to achieving high-throughput query processing. When applied to the RelatedQS workload in isolation using a standard IVF index (*i.e.*, workload-oblivious setting), we obtained a 17× improvement in query processing throughput over the baseline PreFilter in Section 6.2. Furthermore, it is shown in Section 6.3 that our batch query processing technique (Algorithm 3) can provide up to three orders of magnitude runtime improvements compared to issuing queries one at a time. Despite our focus on hybrid queries, our batching strategies can be applied to traditional vector search queries. In particular, Figure 7b shows that vector similarity-based batching can still provide $\approx$ 20× runtime improvements compared to a standard IVF index.

Lastly, although our batching strategies are not applicable in online settings, our workload-aware partitioning strategy can benefit online hybrid query processing systems. We find that when used in isolation (*i.e.*, no batching) workload-aware partitioning and indexing (Section 4) reduces the number of tuples scanned for the RelatedQS workload by 77%, resulting in a 4× faster query processing time compared to PreFilter. This result is reinforced by Figure 7a, which shows that our indexing approach reduces the query processing time of query template T4 by up to 10× even when using a batch size of 1. Given HQI's ability to benefit unseen queries without the need for re-indexing when workload characteristics are stable over time, HQI trained using a representative sample of query logs is still effective in reducing the query processing time.

7 CONCLUSION

This paper describes the design and implementation of a system for batch hybrid vector similarity search query execution that exploits knowledge of the query workload to achieve high throughput. Our design is informed by the characteristics of industrial query workloads over a KG. Specifically, we introduce a workload-aware vector data partitioning scheme to fit the vector search index layout to the query workload and the vector dataset; and describe a multi-query optimization technique to reduce the overhead of vector similarity computations. Experiments using industrial query workloads and a variety of publicly available ANN benchmarks show that the combination of proposed workload-aware index design and query batching strategies provide an order of magnitude higher batch query execution throughput over existing solutions.

REFERENCES

[1] 2023. LLM Powered Search - Vectara. https://vectara.com Accessed on February 28, 2023.
[2] 2023. Vector Database for Vector Search | Pinecone. https://www.pinecone.io Accessed on February 28, 2023.
[3] 2023. Vespa - the big data serving engine. https://vespa.ai Accessed on February 28, 2023.
[4] Manos Athanassoulis, Kenneth S Bøgh, and Stratos Idreos. 2019. Optimal column

混合工作负载的数据布局。《VLDB基金会会刊》12，13（2019），2393–2407。[5] Hongyun Cai、Vincent W Zheng 和 Kevin Chen-Chuan Chang。2018。图嵌入综合综述：问题、技术与应用。《IEEE知识与数据工程汇刊》30，9（2018），1616–1637。[6] Ines Chami、Sami Abu-El-Haija、Bryan Perozzi、Christopher RÃ© 和 Kevin Murphy。2022。《图上的机器学习：模型与综合分类法》。《机器学习研究期刊》23，89（2022），1–64。http://jmlr.org/papers/v23/20-852.html[7] Jialin Ding、Umar Farooq Minhas、Badrish Chandramouli、Chi Wang、Yinan Li、Ying Li、Donald Kossmann、Johannes Gehrke 和 Tim Kraska。2021。面向云分析工作负载的实例优化数据布局。载于《2021年国际数据管理会议论文集》，418–431。[8] Jerome H Friedman、Jon Louis Bentley 和 Raphael Ari Finkel。1977。一种以对数级期望时间查找最佳匹配的算法。《ACM数学软件汇刊》（TOMS）3，3（1977），209–226。[9] Albert Gordo、Jon Almazán、Jerome Revaud 和 Diane Larlus。2016。深度图像检索：学习用于图像搜索的全局表示。载于《欧洲计算机视觉会议》。施普林格，241–257。[10] Mihajlo Grbovic 和 Haibin Cheng。2018。使用嵌入实现爱彼迎搜索排序的实时个性化。载于《第24届ACM SIGKDD知识发现与数据挖掘国际会议论文集》，311–320。[11] Aditya Grover 和 Jure Leskovec。2016。node2vec：面向网络的可扩展特征学习。载于《第22届ACM SIGKDD知识发现与数据挖掘国际会议论文集》，855–864。[12] 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。arXiv预印本 arXiv:2206.13843（2022）。[13] Malay Haldar、Mustafa Abdool、Prashant Ramanathan、Tao Xu、Shulin Yang、Huizhong Duan、Qing Zhang、Nick Barrow-Williams、Bradley C Turnbull、Brendan M Collins 等。2019。将深度学习应用于爱彼迎搜索。载于《第25届ACM SIGKDD知识发现与数据挖掘国际会议论文集》，1927–1935。[14] Will Hamilton、Zhitao Ying 和 Jure Leskovec。2017。大型图上的归纳式表示学习。《神经信息处理系统进展》30（2017）。[15] Helia Hashemi、Aasish Pappu、Mi Tian、Praveen Chandar、Mounia Lalmas 和 Benjamin Carterette。2021。面向音乐和播客的神经即时搜索。载于《第27届ACM SIGKDD知识发现与数据挖掘会议论文集》，2984–2992。[16] Shizhu He、Kang Liu、Guoliang Ji 和 Jun Zhao。2015。使用高斯嵌入学习知识图谱表示。载于《第24届ACM信息与知识管理国际会议论文集》，623–632。[17] Benjamin Hilprecht、Carsten Binnig 和 Uwe Röhm。2020。为云数据库学习分区顾问。载于《2020年ACM SIGMOD数据管理国际会议论文集》，143–157。[18] MD Zakir Hossain、Ferdous Sohel、Mohd Fairuz Shiratuddin 和 Hamid Laga。2019。图像字幕生成深度学习综合综述。《ACM计算综述》（CsUR）51，6（2019），1–36。[19] Ihab F. Ilyas、Theodoros Rekatsinas、Vishnu Konda、Jeffrey Pound、Xiaoguang Qi 和 Mohamed Soliman。2022。Saga：一个以大规模持续构建和提供知识的平台。载于《2022年国际数据管理会议论文集》。 2259 2272 10 1145 3514221 3526049 – 。[20] https://doi.org/ 。/ 。Piotr Indyk 和 Rajeev Motwani。 。近似最近邻：致力于消除维数灾难。载于《第三十届ACM计算理论年度研讨会论文集》，604–613。[21] 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。[22] Hervé Jégou、Romain Tavenard、Matthijs Douze 和 Laurent Amsaleg。2011。搜索十亿个向量：使用源编码进行重排序。载于《2011年IEEE声学、语音与信号处理国际会议（ICASSP）》。IEEE，861–864。[23] Jeff Johnson、Matthijs Douze 和 Hervé Jégou。2017。使用图形处理器进行十亿级相似性搜索。https://doi.org/ 10 48550 1702 08734 。/ ARXIV。 。Tim Kraska。 。面向实例优化数据系统。《VLDB基金会会刊》14，12（2021）。[25] Jie Li、Haifeng Liu、Chuanghua Gui、Jianyu Chen、Zhenyuan Ni、Ning Wang 和 Yuan Chen。2018。京东电商平台实时视觉搜索系统的设计与实现。载于《第19届国际中间件会议产业论坛论文集》，9–16。[26] David C Liu、Stephanie Rogers、Raymond Shiau、Dmitry Kislyuk、Kevin C Ma、Zhigang Zhong、Jenny Liu 和 Yushi Jing。2017。Pinterest上的相关图钉：真实场景推荐系统的演进。载于《第26届万维网国际会议伴随论文集》，583–592。

[27] Zhuoran Liu、Leqi Zou、Xuan Zou、Caihua Wang、Biao Zhang、Da Tang、Bolin Zhu、Yijie Zhu、Peng Wu、Ke Wang 和 Youlong Cheng。2022。Monolith：采用无冲突嵌入表的实时推荐系统。载于第5届在线推荐系统与用户建模研讨会（ORSUM2022），与第16届ACM推荐系统会议同期举行。[28] Yu A Malkov 和 Dmitry A Yashunin。2018。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（2018），824–836。[29] Shervin Minaee、Nal Kalchbrenner、Erik Cambria、Narjes Nikzad、Meysam Chenaghlu 和 Jianfeng Gao。2021。基于深度学习的文本分类：全面综述。《ACM计算综述》（CSUR）54，3（2021），1–40。[30] Jason Mohoney、Roger Waleffe、Henry Xu、Theodoros Rekatsinas 和 Shivaram Venkataraman。2021。Marius：在单台机器上学习大规模图嵌入。载于第15届{USENIX}操作系统设计与实现研讨会（{OSDI} 21）。533–549。[31] Shumpei Okura、Yukihiro Tagami、Shingo Ono 和 Akira Tajima。2017。面向数百万用户的基于嵌入的新闻推荐。载于第23届ACM SIGKDD知识发现与数据挖掘国际会议论文集。1933–1942。[32] Mingdong Ou、Peng Cui、Jian Pei、Ziwei Zhang 和 Wenwu Zhu。2016。保留非对称传递性的图嵌入。载于第22届ACM SIGKDD知识发现与数据挖掘国际会议论文集。1105–1114。[33] Aditya Pal、Chantat Eksombatchai、Yitong Zhou、Bo Zhao、Charles Rosenberg 和 Jure Leskovec。2020。Pinnersage：用于Pinterest推荐的多模态用户嵌入框架。载于第26届ACM SIGKDD知识发现与数据挖掘国际会议论文集。2311–2320。[34] Loïc Paulevé、Hervé Jégou 和 Laurent Amsaleg。2010。局部敏感哈希：哈希函数类型和查询机制的比较。《模式识别快报》31，11（2010），1348–1358。[35] Bryan Perozzi、Rami Al-Rfou 和 Steven Skiena。2014。Deepwalk：在线学习社会表示。载于第20届ACM SIGKDD知识发现与数据挖掘国际会议论文集。701–710。[36] An Qin、Mengbai Xiao、Yongwei Wu、Xinjie Huang 和 Xiaodong Zhang。2021。Mixer：在Web规模下高效理解和检索视觉内容。《VLDB基金会会刊》14，12（2021），2906–2917。[37] P Griffiths Selinger、Morton M Astrahan、Donald D Chamberlin、Raymond A Lorie 和 Thomas G Price。1979。关系数据库管理系统中的访问路径选择。载于1979年ACM SIGMOD国际数据管理会议论文集。23–34。[38] Liwen Sun、Michael J Franklin、Sanjay Krishnan 和 Reynold S Xin。2014。用于积极跳过数据的细粒度分区。载于2014年ACM SIGMOD国际数据管理会议论文集。1115–1126。[39] Jian Tang、Meng Qu、Mingzhe Wang、Ming Zhang、Jun Yan 和 Qiaozhu Mei。2015。LINE：大规模信息网络嵌入。载于第24届万维网国际会议论文集。1067–1077。[40] Roger Waleffe、Jason Mohoney、Theodoros Rekatsinas 和 Shivaram Venkataraman。2023。MariusGNN：图神经网络的资源高效核外训练。载于ACM SIGOPS欧洲计算机系统会议（EuroSys）。[41] Jizhe Wang、Pipei Huang、Huan Zhao、Zhibo Zhang、Binqiang Zhao 和 Dik Lun Lee。2018。阿里巴巴电子商务推荐中的十亿级商品嵌入。载于第24届ACM SIGKDD知识发现与数据挖掘国际会议论文集。839–848。[42] 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。[43] Quan Wang、Zhendong Mao、Bin Wang 和 Li Guo。2017。知识图谱嵌入：方法与应用综述。《IEEE知识与数据工程汇刊》29，12（2017），2724–2743。[44] 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基金会会刊》13，12（2020），3152–3165。[45] Wen Yang、Tao Li、Gai Fang 和 Hong Wei。2020。PASE：PostgreSQL超高维近似最近邻搜索扩展。载于2020年ACM SIGMOD国际数据管理会议论文集。2241–2253。[46] Zongheng Yang、Badrish Chandramouli、Chi Wang、Johannes Gehrke、Yinan Li、Umar Farooq Minhas、Per-Åke Larson、Donald Kossmann 和 Rajeev Acharya。2020。qd树：面向大数据分析的数据布局学习。载于2020年ACM SIGMOD国际数据管理会议论文集。193–208。https://doi.org/ 10 1145 3318464 3389770 。/。Muhan Zhang 和 Yixin Chen。。基于图神经网络的链接预测。《神经信息处理系统进展》31（2018），5165–5175。[47] 2018