LHGstore：一种用于快速更新和分析的内存中学习型图存储

Pengpeng Qiao 东京科学大学 日本东京
peng2qiao@gmail.com

Zhiwei Zhang
北京理工大学
中国北京
zwzhang@bit.edu.cn

Xinzhou Wang
北京理工大学
北京，中国
xzwang@gmail.com

Zhetao Li 暨南大学
广州，中国
liztchina
@gmail.com

Xiaochun Cao
中山大学
中国深圳
caoxiaochun@mail.sysu.edu.cn

曹阳
东京科学大学
日本东京
cao@c.titech.ac.jp

arXiv:2603.11596v1 [cs.DB] 12 Mar 2026

摘要

许多现实世界的应用依赖内存动态图，这些动态图必须高效处理频繁更新，同时支持对不断演变的结构进行低延迟分析。由于更新效率与遍历局部性之间存在权衡，尤其是在度分布高度倾斜的情况下，要同时实现这两个目标仍然颇具挑战。这促使人们设计针对现代多核 CPU 上内存图管理进行优化的图索引方案。我们提出了**LHGstore**，一种感知度的**学**习型**分**层**图**存储，首次将学习型索引整合进图管理中。LHGstore 设计了一个两级层次结构，将顶点访问与边访问解耦，并进一步采用适应顶点度的数据结构组织各顶点的边。低度顶点使用轻量级数组，以最大限度地提升遍历局部性；高度顶点则采用学习型索引，以提高更新吞吐量。大量实验表明，与最先进水平的内存图存储系统相比，LHGstore 的吞吐量提高了 5.9–28.2× 倍，分析速度也显著更快。

1 引言

许多真实世界图 [1, 3] 广泛应用于社交网络、推荐系统和欺诈检测等领域 [15]
这些图会随时间动态演化，而应用程序同时需要进行图分析以提取洞见 [26, 30,35]。例如，在金融网络中，账户是节点，交易是边；随着新交易产生或现有交易被取消，图会动态更新，而广度优先搜索（BFS）等算法则用于追踪可疑的交易路径。因此，设计一种高效的图存储结构，以同时支持（$\mathbf{R}_1$）高吞吐量的图更新（例如，边的插入和删除），以及（$\mathbf{R}_2$）高性能的图分析

允许免费制作本作品的全部或部分内容的数字副本或纸质副本，供个人或课堂使用，但不得为营利或商业目的制作或分发副本，且副本首页必须包含此声明和完整引文。作者以外的其他权利人拥有的本作品组成部分，其版权必须得到尊重。注明出处后，可摘录本作品内容。以其他方式复制、重新出版、发布到服务器或分发到列表，均须事先获得明确许可和/或支付费用。请通过 permissions@acm.org 申请许可。*DAC'26*，美国加利福尼亚州长滩© 2026 版权归所有者/作者所有。出版权授权给 ACM。ACM ISBN 978-1-4503-XXXX-X/2018/06 https://doi.org/XXXXXXX.XXXXXXX

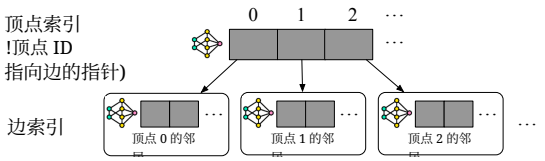


图1：图索引的高级架构。顶点索引记录所有顶点的邻居（nbrs）所在位置，并为每个顶点的边保存一个边索引列表。

（例如，广度优先搜索（BFS）、单源最短路径（SSSP））是一个重要问题。也就是说，图更新时间和图分析时间都必须足够快。

若干现有的内存动态图存储系统 [5,9, 19, 25, 27, 31, 42] 难以同时满足这两项要求。首先，传统索引结构存在固有的低效问题。例如，B+树 [10]或自适应基数树 [18]依赖代价高昂的键比较，因此都不适合图更新。其次，一些系统优先考虑更新效率，却牺牲了遍历局部性，导致分析性能不佳 [6, 40]。例如，类似CSR的布局 [32] 具有出色的分析顺序局部性，但会使动态更新的代价高得难以承受。较新的方案，如 Teseo [19], 、LiveGraph [41], 和 LSMGraph [38] ，通过压缩内存数组、基于日志的存储或 LSM树改善了特定方面，但仍依赖比较

繁重 或需要大量移位的 数据结构，其更新或查找成本仍然相当高。这一差距凸显了对新型索引的需求，以同时优化更新和分析。

权衡。 为动态图设计图索引时，更新效率与遍历局部性之间存在根本性的权衡。高吞吐量更新（$\mathbf{R}_1$）需要高效的查找 和插入 操作，并尽量减少数据移动，因此更适合采用非连续布局。相比之下，高性能分析（$\mathbf{R}_2$，例如广度优先搜索）依赖连续内存布局，以最大限度地提高 CPU缓存局部性。这两种需求本质上相互冲突：非连续布局能够降低更新成本，却会使内存访问变得零散；而连续布局有利于分析，却需要在插入和删除边时进行代价高昂的元素移动。**因此，传统图索引要同时支持动态图中的高效更新和分析并非易事。**

表 1：顶点度数分布。

图	≤ 10邻居	≤ 100邻居	≤ 1000邻居	Avg. 度	Max 度
Twitter	64.5%	95.4%	99.51%	39	2997469
LiveJournal	65%	97.2%	99.98%	17	14815

学习型索引的最新进展 [4, 7, 17, 29, 33, 34] 表明，轻量级机器学习模型可以通过预测驱动的数据布局，取代比较开销较大的 索引。借助键分布，学习型索引能够实现接近$O(1)$ 的查找和插入性能，同时保持有序布局；这些特性恰好适用于现代多核 CPU 上的内存系统，因为在这类系统中，缓存局部性对性能起主导作用。此外，图邻接表也可以自然地建模为键值对，其中键是顶点 ID，值是邻居 ID，因此学习型索引很适合作为动态图存储的候选方案。然而，将学习型索引直接应用于图仍存在固有限制。例如，当高阶顶点的许多边都被预测到同一位置时，查找需要进行额外扫描，插入也可能触发大规模移动，从而导致 $O(deg(u))$ 开销。这种开销使学习型索引无法在动态图中充分发挥其优势。

为了解决上述局限性，我们提出了**LHGstore**，一种度 -感知的**L**earned **H**ierarchical **G**raph 索引框架。如图1所示，LHGstore采用双层分层架构，将顶点查找与邻居访问解耦。这使邻接表能够通过近乎常数时间的预测来获益，同时保留图分析所必需的有序布局。具体而言，我们将图建模为一组键值对，并利用学习型索引 [7] 作为支持分层图存储的核心子模块。LHGstore引入了二级边级索引，为每个顶点配备一个学习型索引，理论上可将查找和插入成本均降低至预期的 $O(1)$，从而满足要求（$\mathbf{R}_1$）。此外，现实世界中的动态图通常具有倾斜的顶点度分布，少数顶点度数较高，而多数顶点度数较低，如表1所示。因此，为每个顶点采用统一的"放之四海而皆准"索引设计并非最优。存储顶点邻居的理想结构取决于图分析的访问模式以及执行图更新的成本。这促使我们采用度感知设计。对于低度顶点，我们使用无序数组，以避免不必要的模型开销，并保留有利于分析的顺序局部性（$\mathbf{R}_2$）。对于高度顶点，我们采用边级学习型索引，以实现高效更新和遍历。总之，本文的贡献如下。

—

全新视角。本文首次尝试引入学习型索引，作为兼顾更新效率和遍历局部性 的解决方案，以探索内存动态图存储。

简单而有效的解决方案。我们提出LHGstore，一种度感知学习型分层图索引结构，它将顶点访问与邻居访问解耦，同时根据顶点度分配不同的邻接布局。该设计利用真实世界图中的偏斜度分布，以最小化更新时的数据移动，并最大化分析时的局部性。

大量实验。我们在合成图和真实世界图上，使用三种工作流和五种图分析算法开展实验，证明我们的方法能够高效地处理图更新与分析性能。

2 相关工作

图存储。 一种经典的图存储结构是 CSR [11, 36]，它通过连续存储顶点和边，为静态图提供高分析性能。LLAMA [23] 利用多版本快照，将类似 CSR 的结构扩展到动态图场景。Teseo [19] 采用基于PMA的存储并支持事务，Aspen [6] 利用写时复制PAM树优化读取，LiveGraph [41] 采用 TEL块以实现高效插入，Sortledton [9] 则将哈希表与跳表结合，在更新和查询之间取得平衡。然而，现有方法主要依赖传统索引结构来存储动态图数据，往往面临显著的性能瓶颈。

学习型存储。基于机器学习的存储 [17] 已成为一个颇具前景的研究方向，并且已有多项利用机器学习技术的理论研究 [8, 12, 16, 20–22, 24, 37, 39] 。 [17] 提出了递归模型索引，利用少量层级的模型对数据集进行划分。Ding等人 [7] 提出了 ALEX，这是首个支持动态插入的学习型索引，并引入了一种自顶向下的自适应方法来构建树形线性学习型索引。然而，目前尚无针对图的相关研究。

3 动机与基准

3.1 动机

3.1.1 内存访问模式。 众所周知，图分析算法受内存访问限制 [9]。大多数图算法，包括 Graphalytics [13]，中基准测试的算法，都遵循顺序访问模式或随机访问模式。算法 1 展示了顺序访问模式：算法遍历所有顶点 V，访问它们的邻接表，并根据邻居更新顶点状态。这种模式常见于 PageRank 等迭代算法。相比之下，随机访问模式由运行时决定。

算法 1： 顺序访问模式

```
1 当条件 时
2     遍历 v ∈ V 执行
3         对每个 u ∈ v.nbrs 执行
4             state[v] ← f_update(state[v], v, u);
```

在内存图引擎中，这种顺序模式的性能几乎完全取决于底层存储布局。如果某个顶点的边未连续存放，就会丧失遍历局部性，导致频繁的缓存未命中。对于高度顶点，这个问题尤为严重，因为碎片化内存布局会加剧随机 DRAM访问。优化存储以更好地匹配访问模式，对于加速分析至关重要。

3.1.2 典型图存储。典型的内存图存储 [41]通常采用基于顶点表的架构，由哈希映射、静态顶点表和邻接表组成（图 2）。虽然

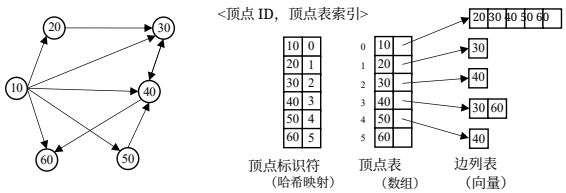


图2：一种典型的图存储。

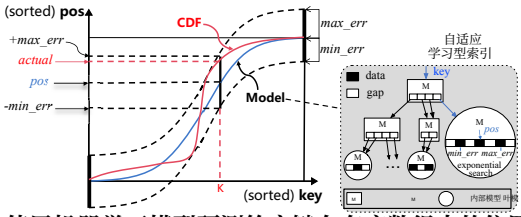


图3：使用机器学习模型预测给定键在有序数组中的位置示例。

这种结构支持 $O(1)$ 顶点查找，但在动态图场景下存在两个关键局限。首先，静态顶点表限制了可扩展性，并使动态顶点插入或删除操作更加复杂。其次，邻接表缺少内部索引，因此边的查找和插入会退化为代价高昂的线性扫描。总之，这些问题使此类设计越来越无法满足高性能内存动态图工作负载的需求，因为随机指针追逐和分散的邻接表会带来难以承受的开销。

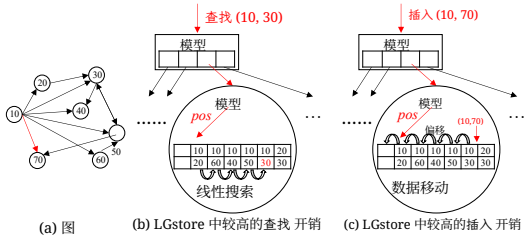
弥合权衡。 基于上述动机，动态图存储面临的核心问题是如何平衡两项相互竞争的需求：**(R1)** 高效更新和 **(R2)** 保持局部性的遍历。连续布局可最大限度地降低分析的访问开销，但更新时会产生高昂的数据移动成本。非连续布局虽能降低更新成本，却会破坏局部性，导致遍历性能不佳。因此，设计一种同时支持动态图快速更新和分析的索引颇具挑战。**我们的研究重点。** 本文旨在正面应对这一矛盾，探索学习型索引和混合数据布局策略。

3.2 我们的基线设计

定义 1. 图到键值对。 图 $G = (V, E)$ 是顶点集合 V和边集合 E。我们用 $|V|$表示顶点数，用 $|E|$表示边数，用 $v \in V$ $deg(v)$ (u, v) $u\ v \in V$ 表示顶点的度。每条边都是一个键值对，其中 ，。例如，对于每条边 (u, v)，u是键， v是值。每个顶点 $v \in V$由一个小于 $|V|$ 的唯一非负整数表示（即 $v \in \{1, 2, ..., |V|\}$） 。

我们的基线设计：学习型图存储（LGstore）。LGstore 的核心思想是利用机器学习模型 [7] 学习图的分布。根据定义 1，图数据可以看作一组键值对。例如，对于每条边 (u, v)， u 是键， v是值。LGStore 的结构源自自适应学习索引 [7], ，如图 3 所示。LGStore 是一种递归模型索引（RMI [14]），其中所有模型都组织成树结构。每个非叶节点（称为内部节点）都包含一个学习模型和指针

指向其子节点。每当需要确定给定键（顶点）的位置时，内部节点都会使用其模型来确定需要搜索的子节点。事实上，每个内部节点



(a) 图 (b) LGstore 中较高的查找 开销 (c) LGstore 中较高的插入 开销

图 4：LGstore 中叶节点的示例。

管理一定范围内的顶点，其模型会将所有顶点划分为若干分区，分区数量与其子节点数量相同。每个叶节点 负责管理边，同时保存一个学习模型，用于将顶点映射到一个位置。叶节点使用间隔数组 [2] 存储每个顶点的边。

示例 *3.1.* ***LGstore*** 存储图。 我们使用 LGStore 存储动态图，如图 4 所示。顶点 10 的所有边都以未排序的方式连续存储在一个叶节点中，如图 4(b) 所示。当我们查找边 $(10, 30)$ 时，叶节点的线性模型会预测 $(10, 20)$ 的位置，因此我们必须对 5 个元素进行线性搜索。如图 4(c) 所示，当我们插入一条新边 $(10, 70)$ 时，需要移动 5 个元素，才能将这条新边放到正确的位置。

4 LHGstore设计 4.1 LGstore的局限性

尽管 LGstore 是将学习型索引应用于图的一项自然的初步尝试，但它存在两个局限性。

限制***1***： $O(deg(u))$ **查找和插入的开销。** 如示例 3.1所示，当同一顶点的多条边被预测到同一位置时，它们会紧密存储在一起，中间没有空隙。在这种情况下，查找一条边需要线性扫描该顶点的所有邻接边，而插入一条新边则需要移动多个元素。对于顶点 u，查找 和插入 的理论最坏情况时间复杂度均为 $O(deg(u))$。对于内存环境中的高度顶点，这种 O(deg(u)) 的行为会导致漫长的缓存行遍历和 DRAM突发访问，严重限制吞吐量。

限制***2***：**偏斜度分布下的一刀切布局。** 表 1 展示了具有偏斜性的真实世界图中顶点度的分布。面对偏斜图，LGstore采用“一刀切”的数据结构设计，在处理和更新偏斜图时无法充分发挥性能。对于度较低的顶点，简单的数据结构只需少量间接访问，并支持高效遍历和更新。然而，对于度较高的顶点，采用渐近搜索和更新性能更优的复杂数据结构可能更合适。因此，要获得更好的性能，有必要区别对待低度和高度顶点。

4.2 LHGstore 概述

我们提出了 LHGstore，以解决 LGstore 中的上述问题。LHGstore 的架构如图5所示。

层次化设计。 为克服 局限-*1*，LHGstore 采用两级层次索引，以消除模型预测中相同位置带来的影响。在第一级（顶点索引）中，LHGstore 使用学习型索引存储每个顶点的边，但

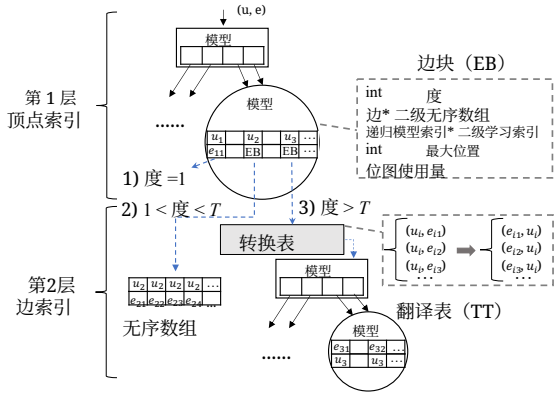


图5：LHGstore的结构。

仅当其度=1。否则，每个顶点的值（度>1）是一个边块，指向相应的第二级索引（边索引）。在第二级中，每个边索引采用另一个学习型索引来存储顶点的所有边，通过间隔数组确保有序邻接和局部性，并实现高效更新。该设计将顶点 u 的查找和插入成本从 LGstore 中的 $O(deg(u))$ 降低至 LHGstore 中的预期 $O(1)$。

度感知设计。 为解决限制*2*，LHGstore采用了一种混合策略，使顶点的布局取决于该顶点的度。对于不同度的顶点，基于数组和基于学习的索引结构在可更新性和局部性方面提供不同的保证，并存在交叉点。我们使用度阈值 T 来区分这两种布局（交叉点的分析见第4.4节）。

4.3 顶点索引

如图5所示，我们使用 LGstore 中引入的自适应学习索引作为顶点级索引。

边块。 与顶点 u_i 对应的边块存储了 u_i的度、两个指向边索引的指针，分别称为二级_未排序_数组 和 二级_学习型_索引，以及边索引的两个辅助变量，分别称为 最大_位置 和 已用。边块按照顶点索引排序。

令 $\{e_{i1}, e_{i2}, ...\}$ 表示 u_i的邻接顶点。当 $deg(u_i)$ 等于 1 时（例如 u_1），一级索引直接在原位存储其邻接顶点（例如e_{11}），无需二级索引。当$deg(u_i)$ 大于 1 时（例如 u_2，u_3），我们使用 边块 存储 u_i所需的元数据。

对于顶点索引，键是顶点 ID u_i，值是指向相应邻接块（边索引）或其边 e_i的指针。当插入或删除顶点 u_i 的一条边时，我们首先更新顶点 u_i 的度，然后在二级索引中执行相应操作。

4.4 边索引

平衡局部性与可更新性。 基于数组和基于学习型索引的数据结构，在局部性和可更新性方面提供了不同的保证。学习型索引更新迅速，因此在图更新方面表现良好。相比之下，基于数组的结构可更新性不如学习型索引，但由于边

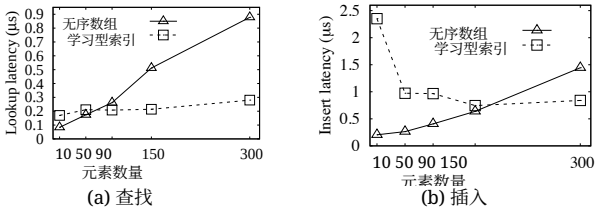


图6：无序数组和学习型索引上查找与插入操作的延迟。

在内存中连续存储，具有更好的数据局部性，因此能够支持快速的图分析算法。事实上，学习型索引和数组的更新性能会随着顶点的度而变化，并存在一个交叉点。为验证这一点，我们进行了微基准测试，比较数组和学习型索引的查找与插入延迟。图6所示的结果揭示了一个明确的交叉点：对于 $n < 70$，无序数组中的查找速度快于学习型索引；对于 $n < 150$，无序数组中的插入速度则超过学习型索引。这一交叉点为选择索引阈值提供了有依据的指导原则 T，该阈值决定每个顶点的边是存储在无序数组中还是学习型索引中。

对于每个高阶顶点（$d(u) > T$），LHGstore 为每个顶点分配一个学习型索引作为边索引，以确保高效的查找和插入。如果直接将顶点 ID u_i 用作 u_i所有边的键，则所有键值对都具有相同的键，模型会预测相同的位置，从而使学习型索引失效。为解决这一问题，我们引入转换表，将邻居 ID e_i 视为键，将 u_i 视为值。例如，我们首先通过转换表将所有边 $\{(u_i, e_{i1}), (u_i, e_{i2}), ...\}$ 转换为$\{(e_{i1}, u_i), (e_{i2}, u_i), ...\}$，然后构建学习型索引。对于每个低阶顶点（$d(u) \leq T$），则采用无序数组，它支持常数时间更新，并保留用于分析的顺序扫描。所有学习型索引模块均使用自适应线性回归模型 [7], 实现，这些模型可以高效训练，开销可忽略不计。

示例 *4.1.* ***LHGstore*** 存储图。 图5 展示了 LHGstore 如何在动态图中存储度数不同的顶点。考虑图5(a)所示的图。我们假设阈值为 $T = 5$。顶点 v_{20} 的度为 1，因此其唯一的邻居直接存储在顶点索引中，无需边级结构。顶点 v_{30} 的度为 2（$1 < deg(v_{30}) < T$），其邻居存储在无序数组中，通过将新邻居放入空闲槽位来支持高效插入。顶点 10 有 5 条边 $\{v_{20}, v_{30}, v_{40}, v_{50}, v_{60}\}$。由于 $deg(v_{10}) > T$，其边块由边级学习索引管理。例如，插入一条新边 (v_{10}, v_{70})时，模型会预测其近似位置（70 之后），再通过少量校正确定最终插入位置，从而将插入复杂度降低至期望近似 $O(1)$。

4.5 操作

findEdge(u, v)。算法 2 展示了*findEdge*(u, v) 的过程。findEedge(u, v) 返回图中是否存在边 (u, v)。给定一条边 (u, v)，LHGstore 首先使用顶点索引（VIndex）预测 u 所属的位置，然后获取

算法 2： *findEdge*(u, v)

```
1  位置(u) ← VIndex.predict(u, v) ;
2  如果 pos(u).value == null ，则
3      返回 False            /* 不存在的顶点 u */
4  否则如果 位置(u).value == v 则
5      返回 True             /* deg(u) = 1 */
6  否则
7      边块(u) = 位置(u).值;      /* deg(u) > 1 */
8      若边块(u).degree > T 则
9          EIndex = 边块(u).sec_learned_index ;
10         (v, u) = 翻译_表(u, v);
11         value ← EIndex.predict(v, u).value ;
12     else
13         value ← 边块(u)。unsorted_array。遍历;
14     如果 value == null 则
15         返回 假            /* 不存在边 (u, v) */
16     else
17         返回 真
```

边 u的有效载荷，记作 $pos(u)$ （第 1 行）。LHGstore 检查 u 是否存在（第 2 行），或边 (u, v) 是否存储在 VIndex 中（第 4 行）。如果 u 的邻接边存储在边块（EB）中，我们使用sec_学习_索引来预测度为 $> T$时 v的位置（第 7–11 行）。否则，我们遍历 $0 \sim EB(u).max_pos$ 范围内的无序数组，以检查 v是否存在（第 12–17 行）。

算法3： 插入边(u, v)

```
1  位置(u) ← VIndex.predict(u, v);
2  如果 pos(u).value == null 则
3      将 (u, v) 插入 VIndex ;     /* 不存在顶点 u */
4  否则如果 pos(u).值不是 边块 则
5      构造 边块(u);
6      边块(u).度 = 2 ;
7      如果 T == 2则
8          边块(u).sec_learned_index.插入(所有边);
9      否则如果 T > 2 则
10         边块(u).sec_unsorted_array.插入(所有边);
11         更新 边块(u).used,边块(u).max_pos ;
12     将 (u,边块(u)) 更新到 VIndex 中;
13 否则
14     边块(u).degree + 1 ;
15     如果 边块(u).degree ≥ T 则
16         边块(u).sec_learned_index.插入(u, v);
17     否则如果 边块(u).degree < T 则
18         边块(u).sec_unsorted_array.插入(u, v);
19         更新边块(u).used，边块(u).max_pos ;
```

表2：图统计信息。

图	# 顶点	# 边	最大度	平均度
G500-24	8,870,942	260,379,521	406,416	58
G500-26	32,804,978	1,051,922,853	1,003,338	54
Orkut	3,072,441	117,185,083	33,007	43
LiveJ	3,997,962	34,681,189	14,815	17

插入边(u, v). 在算法3中，插入边(u, v) 会将一条边 (u, v) 插入图中。与 $findEdege(u, v)$类似，LHGstore 首先从 VIndex 获取位置(u)（第1行）。如果 u不存在，则直接将其作为一条新边 (u, v) 插入 VIndex（第3行）。如果 边块(u) 不存在（即 u的度等于 1），LHGstore 需要使用 $deg(u)$=2构建边块(u)，并根据 T 的大小决定是否将 u 的边索引构建为 $sec_learned_index$ 或 $sec_unsorted_array$（第4至11行）。例如，如果 $T = 2$，则使用学习型索引构建边块(u)（第8行）；否则，使用无序数组构建（第10至11行）。随后，LHGstore 将边块 边块(u) 更新到 VIndex 中（第12行）。如果边块(u) 存在，将 (u, v) 插入现有边块时有两种情况：(1) 当 边块(u).度大于或等于 T时，LHGstore 使用 $sec_learned_index$存储边 (u, v)（第16行）。(2) 如果 u的度小于 T，则 LHGstore 将 (u, v) 插入无序数组的第一个空位，并更新 $EB(u).used$和 $EB(u).max_pos$（第18至19行）。

删除边(u, v)。 删除边(u, v) 的过程与插入操作类似。LHGstore 首先获取 (u, v) 的位置，然后从 VIndex、$sec_learned_index$ 或 $sec_unsorted_array$ 中删除边 (u, v) 。为确保删除操作的效率，即使顶点的度小于 T，我们也不会将 $sec_learned_index$ 转换为 $sec_unsorted_array$。

5 实验

5.1 实验设置

运行环境。我们的系统使用 C++ 实现，所有实验均在配备 2.1GHz Intel Xeon Silver 4110 CPU 和 256G RAM 的 Linux 系统上进行。默认情况下，我们设置 $T = 60$ 。

对比系统。 我们在六种系统上进行评估：***Teseo***[19],***Sortledton*** *[9]*, ***LiveGraph*** *[41]*,***Aspen*** *[6]*,***LSGraph*** *[28]*,和***LGstore***。

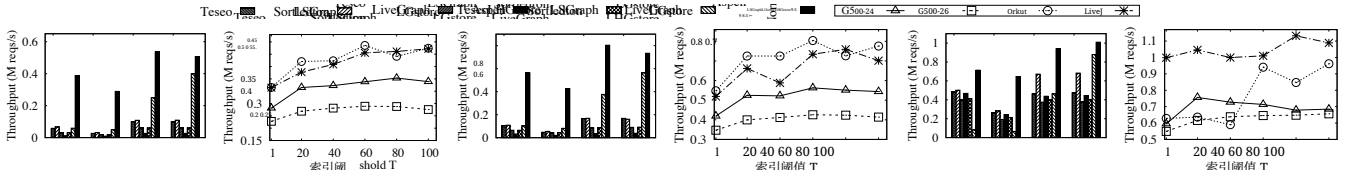
数据集。我们使用两个合成数据集 Graph500-24（G500-24）和 Graph500-26（G500-26）（来自 LDBC Graphalytics基准测试）[13], ，以及两个真实世界数据集 Orkut 和 LiveJournal（LiveJ）进行评估。表2展示了这些数据集的统计信息。

事务型工作负载。我们评估三种工作负载的吞吐量：仅写入（A）、50% 写入和读取（B），以及仅读取（C）。**图分析算法**。我们评估了 LDBC Graphalytics基准测试 [13] 中的五种图分析算法。

5.2 更新吞吐量

Fig.

图 7(a)、7(c) 和 7(e) 报告了三种工作负载下的吞吐量。在所有数据集上，LHGstore 在这三种设置下均实现了最高吞吐量。在工作负载 A 下，LHGstore 相较于 LGstore，在 G500-24、G500-26、Orkut 和 LiveJ 上分别提升了 6.6×、5.9×、2.2× 和 2.2×，比最慢的基准方法最高提升 12.1–28.2×。在工作负载 B 下，LHGstore 的吞吐量比 LGstore 高 1.3–5.4×，比依赖跳表、日志或版本化结构的系统最高高出 9.3–20.1×。即使在局部性发挥更重要作用的工作负载 C 下，LHGstore

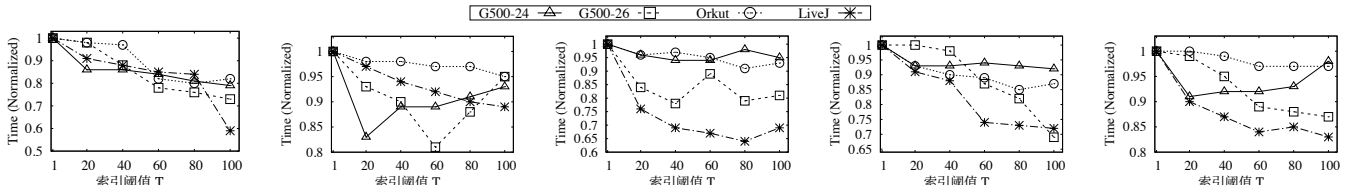


(a) 工作负载 A (b) Workloa d A (c) 工作负载 B (d) 工作负载 B (e) 工作负载 C (f) 工作负载 C

图 7：工作负载 A、B 和 C 下的吞吐量对比，以及改变.....的影响 T.

表 3：四个数据集上五种图算法运行时间的对比。

系统	BFS				PageRank				LCC				WCC				SSSP			
	LiveJ	Orkut	G500-24	G500-26	LiveJ	Orkut	G500-24	G500-26	LiveJ	Orkut	G500-24	G500-26	LiveJ	Orkut	G500-24	G500-26	LiveJ	Orkut	G500-24	G500-26
Teseo	48.7 秒	286.1 秒	586.4 秒	3091.2秒	294.9秒	1909.6秒	3904.8秒	19297.9秒	0.57秒	11.4秒	212.3秒	1216.3秒	55.9秒	287.9秒	614.1秒	3133.6秒	204.2秒	1413.1秒	3062.3秒	15550.2秒
Sortledton	25.1秒	341.6秒	562.3秒	2984.9秒	498.2秒	2827.3秒	4792.6秒	28443.5秒	0.65秒	11.4秒	215.0秒	1250.0秒	65.2秒	211.7秒	673.2秒	3410.5秒	152.4秒	1139.7秒	2591.2秒	12249.8秒
LiveGraph	69.2秒	823.4秒	987.3秒	5630.7秒	997.4秒	4725.8秒	6015.8秒	41781.3秒	0.82秒	16.5秒	254.7秒	1565.3秒	110.5秒	544.1秒	1123.2秒	5922.7s	298.7s	2397.4s	3577.9s	19441.2s
Aspen	24.5秒	278.2秒	548.7秒	2967.6秒	517.3秒	2180.6秒	4438.2秒	30028.8秒	0.70秒	13.1秒	230.3秒	1309.2秒	81.3秒	197.3秒	635.4秒	3045.5秒	233.1秒	1299.3秒	2824.5秒	13912.8秒
LSGraph	22.8秒	273.2秒	540.5秒	2915.5秒	388.2秒	2279.1秒	4297.7秒	24883.1秒	0.69秒	16.5秒	254.7秒	1656.3秒	56.8秒	272.8秒	601.2秒	3008.2秒	97.8秒	951.4秒	2498.2秒	12773.4秒
LGstore	12.1秒	29.4秒	105.0秒	622.5秒	112.6秒	297.9秒	1212.1秒	7834.1秒	0.67秒	17.3秒	1661.9秒	15977.1秒	13.9秒	28.8秒	119.4秒	681.1秒	47.2秒	130.3秒	459.6秒	3302.4秒
LHGstore	**8.8s**	**24.5秒**	**106.1秒**	**619.2秒**	**73.1秒**	**204.8秒**	**805.4秒**	**5164.5秒**	**0.27秒**	**4.9s**	**109.9秒**	**521.1秒**	**11.1秒**	**26.7秒**	**114.3秒**	**577.7秒**	**39.6秒**	**125.9秒**	**434.9秒**	**2913.5秒**



(a) 广度优先搜索 (b) PageRank (c) 局部聚类系数 (d) 弱连通分量 (e) 单源最短路径

图 8：五种算法在不同 T 下的运行时间（相对于 $T = 1$ 时的运行时间进行归一化）。

仍比 LGstore 高 1.1–9.5×，比最慢的基准方法高 5.9–17.4×，这主要是因为学习型边索引避免了对高度顶点进行线性扫描。这些结果表明，LHGstore 在保留对低度顶点进行连续扫描的同时，最大限度地减少了对高度顶点进行昂贵的指针追逐，从而直接提升了真实硬件上的吞吐量。

5.3 分析性能

我们评估了五种图算法的分析性能，如表 3 所示，并根据其主要访问模式进行分组。

随机访问顶点，顺序访问邻居。BFS、弱连通分量和单源最短路径属于此类，其中每次迭代都会访问一个顶点并扫描其邻接表。LHGstore 的性能最佳或接近最佳，相比 Teseo 提速 5.5–11.7× 倍，相比 Sortledton 和 LSGraph 提速 2.6–13.9× 倍。LGstore 凭借其连续布局表现出色，在纯遍历密集型工作负载中有时可接近 LHGstore。然而，与 LGstore 不同，LHGstore 在保持这一性能的同时，并未牺牲更新效率。Teseo 因自适应基数树查找开销高而变慢，而 LiveGraph 和 Aspen 则受指针密集型结构影响，导致局部性下降。

顺序访问顶点和邻居。 PageRank 按顺序扫描所有顶点和邻接表。LHGstore 通过在边块中直接记录顶点度，实现了高效的顺序遍历，因此性能优于所有基准方法。LSGraph 缓解了偏斜问题，但缺乏预测能力；Aspen 和 LiveGraph 则由于采用基于树或基于日志的遍历而产生更高开销。

随机邻居访问。LCC 涉及随机邻居检查。LGstore 依赖线性扫描，性能较差。LHG-store 的学习型索引避免了此类扫描，与 LGstore 相比实现了 2.4–30.6× 的加速比，并优于所有其他基准方法。LiveGraph 和 Aspen 的速度尤其慢，因为随机邻居检查会反复遍历日志或树。

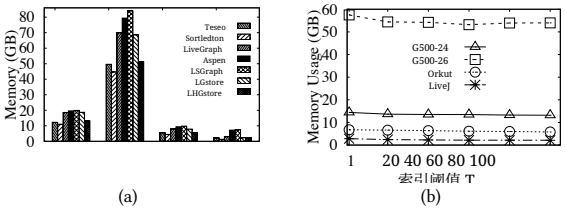
5.4 索引阈值的影响

为评估为不同度的顶点分配不同存储结构所带来的益处，我们考察索引阈值 T 对更新吞吐量和分析性能的影响。如图 7(b)、7(d) 和 7(f) 所示，所有采用 $T > 1$ 的配置都显著优于 T=1，这证实了在偏斜图中区分低度顶点和高度顶点至关重要。在大多数数据集中，吞吐量在 $T = 60$ 附近达到峰值。偶发的波动源于现实世界中的度分布是离散的；当 T 越过一个稠密的度区域时，大量顶点会同时改变布局，从而导致性能发生突变。

对于图分析，图 8(a)–8(e)将运行时间归一化为 $T = 1$ 基准。总体而言，较大的 T 能够提升性能，因为低度顶点在遍历时受益于连续数组。不过，LCC 和 PageRank 等算法的性能并非总是单调提升，因为它们的运行时间既取决于顺序遍历（有利于无序数组），也取决于随机邻接检查（有利于学习型索引）。总体而言，选择合适的T可使 LHGstore 支持高吞吐量的图更新和高性能图分析。

5.5 内存使用

Fig. 图 9(a) 对比了所有系统的内存使用情况。总体而言，LHGstore 的效率具有竞争力，接近 Teseo，且优于大多数其他基准方法。LHGstore 边块中的额外元数据（例如二级索引的指针）会在 G500-26 等大规模图上带来适度开销，但仍小于



(a) (b)

图9：（a）内存使用量对比；（b）LHGstore 在不同 T 下的内存使用量。

LiveGraph 的日志、Aspen 的版本化结构或 Sortledton 的跳表。LGstore 的内存使用量更大，因为它会为每条边重复存储源顶点。此外，不同 T 的内存使用情况如图 9(b) 所示。随着 T 增加，LHGstore 的内存使用量略有下降，因为更多顶点使用无序数组，其空间效率高于学习型索引块。

6 结论

我们提出了 LHGstore，这是一种面向内存动态图的度感知学习型分层图存储框架。通过将顶点访问与边访问解耦，并根据顶点度分配邻接布局，LHGstore 在更新效率与遍历局部性之间实现了权衡。实验表明，与最先进水平的内存图系统相比，LHGstore 在更新吞吐量和分析性能方面均取得了显著提升。

参考文献

[1] Aseem Baranwal、Kimon Fountoulakis和Aukosh Jagannath。2023。稀疏图上消息传递架构的最优性。见*NeurIPS*。40320–40341。[2] Michael A Bender、Martin Farach-Colton和Miguel A Mosteiro。2006。插入排序的复杂度为O (n log n)。*Theory of Computing Systems* 39 (2006)，391–397。[3] Lijun Chang、Mouyi Xu和Darren Strash。2022。面向大型稀疏图的高效最大 k-团簇计算。*VLDB*学报 16, 2 2022 127 139 ( ), – 。[4] Guanduo Chen、Zhenying He、Meng Li和Siqiang Luo。2024。Oasis：一种最优不相交分段学习型范围过滤器。*VLDB*学报 17, 8 (2024)，1911–1924。[5] Zhaole Chu、Zhou Zhang、Peiquan Jin、Xiaoliang Wang、Yongping Luo和Xujian Zhao。2024。LIVAK：面向变长键的高性能内存学习型索引。见第*61*届*ACM/IEEE*设计自动化会议论文集。1–6。[6] Laxman Dhulipala、Guy E Blelloch和Julian Shun。2019。使用压缩纯函数树的低延迟图流处理。见*PLDI*。918–934。[7] Jialin Ding、Umar Farooq Minhas、Jia Yu、Chi Wang、Jaeyoung Do、Yinan Li、Hantian Zhang、Badrish Chandramouli、Johannes Gehrke、Donald Kossmann、David B. Lomet和Tim Kraska。2020。ALEX：一种可更新自适应学习型索引。见*SIGMOD*。ACM，969–984。[8] Paolo Ferragina、Fabrizio Lillo和Giorgio Vinciguerra。2020。为何学习型索引如此有效？见*ICML*。PMLR，3123–3132。[9] Per Fuchs、Jana Giceva和Domagoj Margan。2022。Sortledton：一种通用的事务型图数据结构。*VLDB*学报 15, 6 (2022)，1173–1186。[10] Chuqing Gao、Shreya Ballijepalli和Jianguo Wang。2024。重探B树压缩：一项实验研究。*SIGMOD* 2, 3 (2024)，1–25。[11] Gundolf Haase、Manfred Liebmann和Gernot Plank。2007。面向非结构化稀疏矩阵的希尔伯特序乘法方案。国际并行、涌现与分布式系统杂志 22, 4 (2007)，213–220。[12] Xiao Hu、Yuxi Liu、Haibo Xiu、Pankaj K Agarwal、Debmalya Panigrahi、Sudeepa Roy和Jun Yang。2022。范围查询的选择率函数是可学习的。见*SIGMOD*。959–972。[13] Alexandru Iosup、Tim Hegeman、Wing Lung Ngai、Stijn Heldens、Arnau Prat-Pérez、Thomas Manhardt、Hassan Chafi、Mihai Capota、Narayanan Sundaram、Michael J. Anderson、Ilie Gabriel Tanase、Yinglong Xia、Lifeng Nai和Peter A. Boncz。2016。LDBC Graphalytics：面向并行与分布式平台的大规模图分析基准。*VLDB*学报 9, 13 (2016)，1317–1328。[14] Alon Itai、Alan G Konheim和Michael Rodeh。1981。优先队列的稀疏表实现。见自动机、语言国际研讨会，与编程。Springer，417–431。[15] Arijit Khan、Tianxing Wu和Xi Chen。2024。统一大语言模型+ 知识图谱中的数据管理机遇。见VLDB研讨会。[16] Tim Kraska、Mohammad Alizadeh、Alex Beutel、H Chi、Ani Kristo、Guillaume Leclerc、Samuel Madden、Hongzi Mao和Vikram Nathan。2019。Sagedb：一种学习型数据库系统。见CIDR。[17] Tim Kraska、Alex Beutel、Ed H. Chi、Jeffrey Dean和Neoklis Polyzotis。2018。学习型索引结构的优势。见SIGMOD。ACM，489–504。[18] Viktor Leis、Alfons Kemper和Thomas Neumann。2013。ICDE，Christian S. Jensen、Christopher M. Jermaine和Xiaofang Zhou（编）。38–49。[19] Dean De Leo和Peter A. Boncz。2021。Teseo与结构动态图分析。VLDB学报14, 6 (2021)，1053–1066。[20] Guoliang Li、Ji Sun、Lijie Xu、Shifu Li、Jiang Wang和Wen Nie。2024。Gaussml：端到端数据库内机器学习系统。见ICDE。IEEE，5198–5210。[21] Pengfei Li、Yu Hua、Jingnan Jia和Pengfei Zuo。2021。FINEdex：面向可扩展并发内存系统的细粒度学习型索引方案。VLDB学报15, 2 (2021)，321–334。[22] Lailong Luo、Deke Guo、Richard TB Ma、Ori Rottenstreich和Xueshan Luo。2018。布隆过滤器优化：挑战、解决方案与比较。IEEE通信调查与教程21, 2 (2018)，1912–1949。[23] Peter Macko、Virendra J. Marathe、Daniel W. Margo和Margo I. Seltzer。2015。LLAMA：使用大型多版本数组的高效图分析。见ICDE。363–374。[24] Ryan Marcus、Emily Zhang和Tim Kraska。2020。Cdfshop：探索并优化学习型索引结构。见SIGMOD。2789–2792。[25] Dingheng Mo、Junfeng Liu、Fan Wang和Siqiang Luo。2025。Aster：增强LSM结构以支持可扩展图数据库。SIGMOD 3, 1 (2025)，1–26。[26] Songsong Mo、Yue Zhao、Zhifeng Bao、Quanqing Xu、Chuanhui Yang和Gao Cong。2025。RankPQO：面向参数化查询优化的学习排序。VLDB学报18, 3 (2025年4月)，863–875。[27] Prashant Pandey、Brian Wheatman、Helen Xu和Aydin Buluç。2021。Terrace：面向倾斜动态图的分层图容器。见SIGMOD。ACM，1372–1385。[28] Hao Qi、Yiyang Wu、Ligang He、Yu Zhang、Kang Luo、Minzhi Cai、Hai Jin、Zhan Zhang和Jin Zhao。2024。LSGraph：以局部性为中心的高性能流式图引擎。见EuroSys。33–49。[29] Pengpeng Qiao、Zhiwei Zhang、Yuntong Li、Ye Yuan、Shuliang Wang、Guoren Wang和Jeffrey Xu Yu。2024。AStore：面向支持远程直接内存访问的键值存储的统一自适应学习型索引与缓存。IEEE知识与数据工程汇刊36, 7 (2024)，2877–2894。[30] Scott Sallinen、Juntong Luo和Matei Ripeanu。2023。动态图上的实时PageRank。见HPDC。239–251。[31] Jifan Shi、Biao Wang和Yun Xu。2024。Spruce：一种面向动态图存储的快速且节省空间的结构。SIGMOD 2, 1 (2024)，1–26。[32] Julian Shun和Guy E Blelloch。2013。Ligra：面向共享内存的轻量级图处理框架。见PPoPP。135–146。[33] Zhaoyan Sun、Xuanhe Zhou和Guoliang Li。2023。学习型索引：全面实验评估。VLDB学报16, 8 (2023)，1992–2004。[34] Jiacheng Wu、Yong Zhang、Shimin Chen、Yu Chen、Jin Wang和Chunxiao Xing。2021。具有精确位置的可更新学习型索引。VLDB学报14, 8 (2021)，1276–1288。[35] Shaofeng Yang、Xiandong Liu、Yunting Wang、Xin He和Guangming Tan。2023。大型稀疏图中的快速全点对最短路径算法。见ICS。277–288。[36] Tsun-Yu Yang、Yizou Chen、Yuhong Liang和Ming-Chang Yang。2024。利用按需处理协同优化全外部图计算的可扩展性与效率。ACM存储汇刊21, 2 (2024)，1–31。[37] Yifan Yang和Shimin Chen。2024。LITS：一种面向字符串的优化学习型索引。VLDB学报17, 11 (2024)，3415–3427。[38] Song Yu、Shufeng Gong、Qian Tao、Sijie Shen、Yanfeng Zhang、Wenyuan Yu、Pengxi Liu、Zhixin Zhang、Hongfu Li、Xiaojian Luo等。2024。LSMGraph：采用多级CSR的高性能动态图存储系统。SIGMOD 2, 6 (2024)，1–28。[39] Weiping Yu、Siqiang Luo、Zihao Yu和Gao Cong。2024。CAMAL：通过主动学习优化LSM树。SIGMOD 2, 4 (2024)，1–26。[40] Chao Zhang、Angela Bonifati、Hugo Kapp、Vlad Ioan Haprian和Jean-Pierre Lozi。2023。面向递归标签拼接图查询的可达性索引。见ICDE。IEEE，67–81。[41] Xiaowei Zhu、Guanyu Feng、Marco Serafini、Xiaosong Ma、Jiping Yu、Lei Xie、Ashraf Aboulnaga和Wenguang Chen。2020。LiveGraph：具有纯顺序邻接表扫描的事务型图存储系统。VLDB学报13, 7 (2020)，1020–1034。[42] Xiaowei Zhu、Marco Serafini、Xiaosong Ma、Ashraf Aboulnaga、Wenguang Chen和Guanyu Feng。2020。LiveGraph：具有纯顺序邻接表扫描的事务型图存储系统。VLDB学报13, 7 2020 1020 1034 ( ), – 。