

面向混合负载的图存储系统^①

赵鹏程^{1,2}, 吕 敏^{1,2}

¹(中国科学技术大学 计算机科学与技术学院, 合肥 230026)

²(中国科学技术大学 高性能计算安徽省重点实验室, 合肥 230026)

通信作者: 吕 敏, E-mail: lvmin05@ustc.edu.cn



摘 要: 图在各种应用中扮演着至关重要的角色, 广泛用于建模实体之间的关系. 图面临的工作负载可分为事务型工作负载和分析型工作负载. 许多应用场景需要同时处理这两类工作负载. 然而, 大多数现有的图存储系统只针对其中一种工作负载进行了优化, 无法同时高效地处理两类工作负载. 为了解决这一问题, 本文提出了面向混合工作负载的图存储系统 HGraph. 本文通过仔细分析两类工作负载的访问模式, 设计了一种适应混合工作负载的数据结构. 此外, HGraph 引入了一种基于撤销日志的多版本并发控制实现, 该方案不仅能够节省内存, 还能提升遍历操作的性能. HGraph 还采用了写时复制和乐观并发控制策略, 以优化事务处理流程, 进一步增强系统的并发能力. 在真实和合成数据集上的实验结果表明, HGraph 的性能优于其他图存储系统.

关键词: 图存储; 属性图; 图数据库; 多版本并发控制; 混合事务分析处理

引用格式: 赵鹏程, 吕敏. 面向混合负载的图存储系统. 计算机系统应用. <http://www.c-s-a.org.cn/1003-3254/9895.html>

Graph Storage System for Hybrid Workloads

ZHAO Peng-Cheng^{1,2}, LYU Min^{1,2}

¹(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)

²(Anhui Key Laboratory on High Performance Computing, University of Science and Technology of China, Hefei 230026, China)

Abstract: Graphs play a crucial role in modeling relationships between entities across various applications. Workloads on graphs are typically categorized into transactional and analytical workloads. Many scenarios now require handling both types of workloads simultaneously. However, most existing graph storage systems are optimized for only one type of workload and cannot efficiently handle both simultaneously. In this study, a new graph storage system, HGraph, is proposed to address this issue. A data structure tailored for hybrid workloads is designed through careful analysis of the access patterns of both workload types. In addition, HGraph introduces a multi-version concurrency control (MVCC) implementation based on undo logs, which is memory-efficient and improves traversal performance. HGraph also adopts copy-on-write and optimistic concurrency control strategies to optimize transaction processing, further enhancing system concurrency. Extensive experiments on both real-world and synthetic datasets demonstrate that HGraph outperforms other graph storage systems.

Key words: graph storage; property graph; graph database; multi-version concurrency control (MVCC); hybrid transaction/analytical processing (HTAP)

1 引言

随着技术的不断发展, 针对高度关联实体的建模

需求日益增长, 例如社交网络、生物网络和电子支付等领域^[1]. 图这种数据结构在这些场景中具有天然的优

① 基金项目: 国家自然科学基金面上项目 (62172382)

收稿时间: 2024-12-11; 修改时间: 2025-01-07; 采用时间: 2025-01-24; csa 在线出版时间: 2025-05-12

势,因为图能够直观地将实体表示为顶点,并通过边清晰地表示实体间的关系,因此特别适用于处理具有复杂关系的数据^[2].图存储系统面临的工作负载主要可以分为两类:事务型工作负载和分析型工作负载.

- 事务型工作负载:事务型工作负载通常只涉及轻量级的读写事务,这些事务会不断更新和查询底层的图数据.读事务主要包括访问特定顶点或边的属性,以及执行一跳遍历操作;写事务则是对图进行增量更新,通常仅涉及少量顶点或边的修改.此类工作负载要求图存储系统不仅实现低延迟和高吞吐,还需具备高效的并发控制机制,以支持并发的更新操作^[3].大多数现有研究要么在成熟的关系型数据库或键值数据库基础上扩展图数据管理功能,要么设计专门的图数据库系统,以高效支持此类工作负载.

- 分析型工作负载:分析型工作负载主要聚焦于从图数据中提取有价值的洞察 (insight). 此类工作负载需要在一致的只读快照上运行,并具有以下两个特点: (1) 长时间运行. 图分析任务通常涉及复杂的查询操作,执行时间较长,往往长达数十秒; (2) 大量数据访问. 图分析查询通常从起始顶点出发,遍历多跳邻居,因小世界现象^[4]的影响,读取的数据集可能会变得非常庞大. 许多图分析引擎选择采用应用广泛的压缩稀疏行 (compressed sparse row, CSR) 格式来存储图数据. 在 CSR 格式中,所有顶点和边被存储在两个连续的数组中,这种结构能够最大化地利用空间局部性,从而高效地支持分析型工作负载.

随着对动态图数据进行实时分析的需求日益增加,图存储系统需要能够同时高效地处理事务型和分析型工作负载^[5]. 这一需求在欺诈检测、商业智能和风险分析^[6-8]等应用场景中尤为突出.

现有的图存储系统大多针对某一类工作负载进行优化. 然而,由于事务型工作负载和分析型工作负载对系统的需求存在差异,它们的优化往往存在冲突. 因此,在单一系统中高效地同时支持这两类工作负载成为一项挑战. 传统的解决方案是采用两个独立的存储系统分别处理这两类工作负载,例如使用 Neo4j^[9]处理事务型工作负载,使用 GraphScope^[10]处理分析型工作负载. 然而,这种方式需要在系统间进行频繁的数据传输,通常通过代价高昂且效率低下的抽取-转换-加载 (extract-transform-load, ETL) 过程来实现^[11]. 尽管这种方法能够借助现有的成熟系统,降低开发难度,但仍存在以下

显著缺点: (1) 数据延迟. ETL 过程会引入数据延迟,导致分析型工作负载无法访问最新数据,这对于依赖实时数据的应用 (如欺诈检测) 来说是不可接受的; (2) 效率低下. 两个系统需要维护重复的数据,不仅浪费存储资源,而且 ETL 过程会消耗大量的计算和网络资源,从而导致整体性能下降.

为避免数据重复和高成本的 ETL 过程,一些研究尝试使用单个系统直接处理混合工作负载,例如 SQL-Graph^[12]和 GraphOne^[13]. 然而,这类系统通常难以高效支持邻接链表扫描这一分析型工作负载中的关键操作. 此外,一些系统缺乏良好的并发控制机制,或者直接放弃了事务支持^[13-15],导致在处理并发更新和查询操作时性能显著下降,或由于数据一致性问题产生错误结果. 因此,设计能够同时高效支持事务型和分析型工作负载的图存储系统正变得越来越重要.

本文提出了 HGraph, 一个面向混合工作负载设计的高效图存储系统. HGraph 采用的底层图数据结构确保了邻接链表扫描操作主要通过顺序内存访问完成,从而可以高效处理分析型工作负载. 此外, HGraph 结合了分块和部分排序等优化技术,有效提高了事务型工作负载的处理性能. 为在处理混合工作负载时也能表现出优异的性能, HGraph 设计了一种基于撤销日志 (undo log) 的多版本并发控制机制 (multi-version concurrency control, MVCC), 不仅能够降低内存开销,还能提升遍历操作中版本控制的效率. 此外, HGraph 采用了写时复制和乐观并发控制策略,以优化事务处理流程,进一步增强系统的并发处理能力.

2 研究现状

本节通过分析两类工作负载的特性和需求来阐述 HGraph 设计的动机.

2.1 事务型工作负载

事务型工作负载主要用于管理图数据, 主要包含对顶点或边的创建 (create)、查询 (retrieve)、更新 (update)、删除 (delete) 操作, 即 CRUD 操作. 由于其对低延迟的要求, 确保高效支持这些操作尤为重要. 在这类工作负载中, 边操作通常被认为是当前图存储系统的性能瓶颈, 所以本文主要关注对边操作的优化. 许多应用场景要求执行非盲写 (non-blind write) 操作: 在执行创建操作之前, 需要先检查该边是否已经存在^[16]. 因此, 创建操作以及查询、更新和删除操作在执行时都需要在顶

点的所有邻边中进行特定边的查找,这也是事务型工作负载中的关键操作。

支持事务型工作负载的图存储系统主要可以分为两类。一类是非原生图存储系统,这类系统通常在现有的关系数据库或键值存储之上,增加图数据管理层,来处理事务型工作负载,典型的例子包括 Weaver^[17]、SQL-Graph 和 NebulaGraph^[18],这类系统能够利用底层存储的成熟优化技术,高效地处理事务型工作负载。另一类是原生图存储系统,这类系统从零开始设计,专门用于存储和查询图数据,并针对图数据的特性进行优化,典型的例子包括 Neo4j、TigerGraph 和 Terrace^[19]。其中, Terrace 针对图数据顶点度数呈幂律分布的特性,设计了一种分层的数据结构,以高效存储和访问不同度数的顶点,从而优化了系统性能。为了高效支持事务型工作负载,这类系统通常采用基于跳表或树的索引来快速定位特定的边,从而高效处理负载中的关键操作。

上述图存储系统通常难以高效支持分析型工作负载。例如,基于关系型存储的系统在遍历过程中通常需要执行高成本的连接操作,并会生成大量中间结果,导致性能降低;基于键值存储的系统则存在读放大和随机内存访问的问题,这些因素同样会对性能产生不利影响。此外,一些原生图存储系统采用基于指针的数据结构,在遍历过程中需要频繁进行指针追逐操作,导致性能下降。相关研究表明^[20],顺序内存访问的延迟比随机访问低约 20 倍,比指针追逐低约 150 倍。

2.2 分析型工作负载

分析型工作负载的主要目的是通过执行遍历或复杂图算法,从图数据中提取有价值的信息。以经典的 PageRank 算法(算法 1)为例,本文将说明图算法的典型结构。

算法 1. PageRank 的主循环

数据: *contrib*: 大小为 $|V|$ 的数组,表示某轮每个顶点的贡献; *scores*: 大小为 $|V|$ 的数组,表示下一轮每个顶点的得分。

```

1. for  $v \in V$  do
2.    $incoming \leftarrow 0$ 
3.   for  $e \in v.neighbors$  do
4.      $incoming \leftarrow incoming + contrib[e]$ 
5.    $scores[v] \leftarrow incoming$ 

```

在 PageRank 算法的每次迭代中,需要扫描每个顶点的所有邻边(即遍历),以获取图的拓扑信息,这是算法执行过程中的关键操作。因此,提升遍历操作的性能

对提高算法整体效率至关重要。为此,大多数面向分析型工作负载设计的图分析引擎通常采用 CSR 数据结构。在 CSR 中,每个顶点的所有邻居在内存中被连续存储,从而显著提升了遍历操作的缓存命中率和整体性能。因此,CSR 数据结构能够很好地支持分析型工作负载。然而,由于设计上的限制,CSR 数据结构在更新顶点或边时需要重新构建整个索引结构,会导致较高的时间开销^[21]。因此,CSR 数据结构难以用于处理事务型工作负载。为了解决这一问题,一些研究对 CSR 数据结构进行了改进。例如, PMA (packed memory array)^[22] 通过在数组中预留间隙(gap)以支持动态更新操作。然而,当间隙数量变少或分布不均匀时, PMA 需要执行全局重新平衡操作,这会引入较高的延迟。在后续研究中,尽管有些系统未使用 CSR 数据结构,而是选择了更适合动态更新的邻接链表,但为了高效支持遍历操作,这些系统都会借鉴 CSR 数据结构的设计思路,对邻接链表进行优化,即通过顺序存储顶点的邻边来提升空间局部性,典型的例子包括 LiveGraph、Spruce 和 Chronos^[23-25]。然而,这些优化通常会影响定位特定边操作的效率。例如, LiveGraph 引入了名为事务型边日志(transactional edge log, TEL)的新型数据结构,该结构能够支持高效的顺序扫描操作,然而在定位特定边时, LiveGraph 需要遍历顶点的所有邻边,导致操作效率下降。

2.3 多版本并发控制

设计一个能够高效处理混合工作负载的图存储系统,要求系统不仅能够分别高效支持两类负载,还需在并发处理两类工作负载时保持良好的性能。

分析型工作负载要求在任务执行时图数据保持不变,以确保结果的准确性;而事务型工作负载则会不断修改底层图数据。这两类工作负载在隔离机制上的需求存在冲突。部分研究只提供并行化支持,如 Kineograph^[26]和 TgStore^[27]。这些系统通过定期批量应用更新并生成快照的方式实现动态图分析,然而,由于图算法无法实时访问最新数据,这种方法难以满足实时性要求较高的场景。一些研究引入了简易的并发控制机制以支持并发操作。但由于并发控制机制位于每个操作的关键路径上,其设计会直接影响系统性能。例如, Grace^[28]支持事务性更新,但采用了一种高开销的写时复制(copy-on-write, COW)策略,即每次修改都会将整个邻接链表复制到边日志的尾部。这种策略显著增加

了更新操作的成本,尤其在处理高度数顶点时,性能下降尤为明显.在后续研究中,一些系统选择采用基于锁的隔离机制.在这种机制下,更新事务会因读事务长时间占用数据对象而被阻塞,与更新事务低延迟的需求冲突.MVCC是面对这一矛盾的常用解决方案.MVCC通过快照隔离的方式,实现读写操作的并发执行,同时确保操作语义的正确性^[29].由于长时间运行的读事务可以访问旧版本数据,因此无需对数据加锁,从而显著提升了系统的并发能力.

传统的MVCC通常基于版本链实现.例如,在Sortledton、Teseo和HyPer^[30-32]系统中,每个数据项均关联一个版本链,用于记录修改历史.版本链中的每个条目包含两个时间戳,用以标识修改的可见时间范围,从而使读事务能够通过遍历版本链访问其可见版本.然而,这种实现方式存在以下两方面不足.首先,MVCC需要为每个数据项分配一个指针,用于维护版本链.由于系统中实际会被修改的数据仅占少量比例,绝大部分指针处于闲置状态,导致内存资源的浪费.其次,版本链在查询旧版本数据时会引入大量指针追逐操作,这种内存访问模式会严重影响遍历操作的性能.因此,如何在实现MVCC的同时优化其内存使用和访问性能,已成为一项重要的研究挑战.

3 系统设计

本节将详细介绍HGraph的数据结构设计.与Neo4j和Grasper^[33]相同,HGraph采用了属性图模型,以体现图数据管理的通用性.为实现高效的遍历操作,HGraph设计了一种数据结构,确保遍历操作主要依赖顺序内存访问,从而充分利用缓存带来的性能优势.此外,HGraph将拓扑和属性信息分离存储,以减少非必要的数据库访问,从而降低数据库访问延迟.考虑到高度顶点的管理通常是系统性能的瓶颈,HGraph针对高度顶点的数据库结构进行了优化,以降低其处理事务型工作负载的开销,同时尽量减少对分析型工作负载处理性能的影响.HGraph还设计了适应顶点度数分布的分层数据结构,可以有效缓解内存碎片化问题.此外,为了实现高效的版本控制,HGraph使用撤销日志存储边的历史版本.与传统实现相比,这种新型实现既降低了内存开销,又提升了系统遍历操作的性能.

3.1 数据布局

HGraph的存储主要分为3个部分:顶点、邻接链

表和版本存储.

- 顶点: 顶点更新操作通常较少,并且大多数事务仅访问顶点的最新版本.针对这一特点,HGraph简单地使用一个数组存储顶点属性,并通过版本链管理顶点的历史版本.最新版本可以通过顶点ID直接定位,旧版本则可通过版本链进行访问.

- 邻接链表: 在HGraph中,由于其采用了属性图模型,每条边都具有特定的标签(label),例如社交网络中的“朋友关系”.为提高数据访问和处理效率,HGraph将同一顶点的边根据标签分配到独立的邻接链表中.如图1所示,HGraph在顶点索引和邻接链表之间增加了一层间接索引,称为标签索引块(label index block),用于将不同标签的边进行分离存储.通过顶点ID和标签ID,系统可快速定位到顶点索引和相应标签索引块,从而获取对应邻接链表的地址.

邻接链表的实现方式包括B+树和跳表.HGraph选择了一种改进版本的跳表作为其实现方式,主要原因在于跳表无需全局重新平衡,具有更高的维护效率.具体而言,HGraph将边顺序存储在边块(edge block)中,并通过跳表结构对这些边块进行组织和管理.

关于边块的设计,为提升遍历操作的效率,HGraph充分利用顺序内存访问的优势,将所有邻边尽可能连续地存储在内存中.如图1,每个边块由头部(header)、边单元(edge cell)和边属性3部分组成.其中,头部用于存储边块的元信息,包括边块大小和边的数量等;边单元从左到右依次追加,用于表示边的拓扑信息;而边属性则单独存储在边块的右端,与边单元分离并一一对应.这种设计能够优化系统性能,因为许多工作负载在处理过程中无需访问边属性,从而减少了不必要的内存访问.每个边单元包含目的顶点ID和对应的属性大小.当需要访问特定边 (V_E, V_D) 的属性时,系统会通过扫描边块定位该边对应的边单元.在扫描过程中,系统会累加已扫描边单元的属性大小,从而确定边 (V_E, V_D) 属性的起始位置.

- 版本存储: HGraph使用撤销日志实现边的多版本存储,以节省存储空间并提升遍历性能.为进一步优化内存使用,HGraph将撤销日志的分配粒度设为以顶点为单位,而非为每个标签分别分配.每个顶点的撤销日志记录了与其关联的所有边的更新操作.与传统实现中为每个数据项分配指针以管理其历史版本相比,该方案显著减少了指针引入的内存开销.

撤销日志由多个日志条目和其对应的旧版本数据

组成, 每个日志条目对应一次边的更新操作. 日志条目包含以下信息: 目的顶点 ID、标签 ID、提交时间戳 (表示更新的起始可见时间) 以及属性大小 (表示该边旧版本属性的的大小). 例如, 在图 1 中, 撤销日志末尾的条目(1,0,- T_S ,4)表示标签 ID 为 0 的边(0,1)被未提交

的写事务 (事务 ID 为 T_S) 修改, 修改前该边的属性大小为 4, 对应的旧版本数据为“Gary”. 此外, 撤销日志采用环形结构设计, 当写入操作到达日志尾部时, 系统会从日志头部继续写入, 从而循环利用存储空间, 提高存储效率.

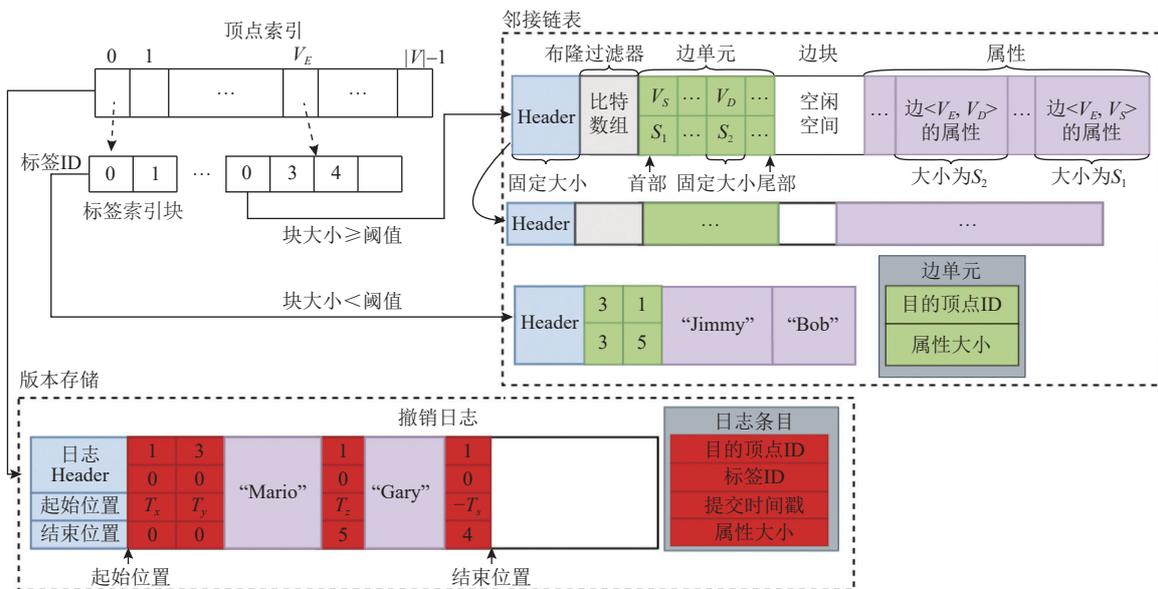


图 1 HGraph 的底层数据结构

当日志条目的提交时间戳小于当前所有活跃事务的时间戳时, 该条目即可被判定为失效, 因为后续操作将不再访问该条目. HGraph 通过修改撤销日志头部中的起始位置来实现垃圾回收, 从而释放失效条目占用的存储空间. 当撤销日志中的所有条目均被回收后, 其占用的存储空间将被归还至内存池, 实现完全回收. 该方法确保了仅有需要版本控制的数据会消耗额外的存储空间, 显著提升了系统的内存利用效率.

3.2 针对高度顶点的优化

高度顶点对系统性能至关重要, 不仅因其在图分析任务中扮演关键角色, 还因为大部分更新操作都与其相关. 然而, 与高度顶点相关的操作通常涉及大量数据访问, 尤其在遍历和更新过程中, 这些操作会引起较大的计算开销, 成为系统性能瓶颈. 因此, 本节将介绍 HGraph 针对高度顶点所采用的优化策略.

- 利用布隆过滤器检查边的存在性. 事务型工作负载中的插入操作在执行时需判断目标边是否存在. 然而, 对于高度顶点, 如果每次插入都需要扫描该顶点的所有邻边, 即使采用最优的追加写方法, 性能也会显

著下降. 为了解决这一问题, HGraph 引入了内存占用小且判断速度快的布隆过滤器, 以高效判断指定边是否存在. 只有当布隆过滤器判断边可能存在时, 系统才会进一步扫描边块, 从而有效减少不必要的扫描操作. 大多数情况下, 系统可以在常数时间内完成插入操作, 大大提升了系统的整体性能.

- 分块策略以减少数据迁移开销. 在删除操作中, 为避免存储空间碎片化, 通常需要进行数据迁移. 然而, 对于高度顶点, 数据迁移的开销通常是难以接受的. 为此, HGraph 采用分块策略, 通过减少迁移数据量, 有效降低数据迁移的开销. 具体而言, HGraph 将高度顶点的所有邻边进行分组, 分别存储在多个相同大小的边块中, 并将这些边块组织为链表. 通过这一设计, 每次删除操作所涉及的数据迁移范围从整个邻接链表缩减为单个边块, 可以有效降低数据迁移的开销. 然而, 该方法引入了少量的指针追逐操作, 会稍微牺牲遍历操作的性能. 为进一步降低迁移操作的频率, HGraph 还采用了懒惰处理策略. 在多数情况下, 系统仅标记被删除的边, 只有当某个边块内标记删除的边数达到预

设阈值时,才会实际移除这些边.这样可以有效减少数据迁移的触发次数,并支持批量处理删除操作,从而提升删除操作的摊销性能.该策略在性能和资源利用之间实现了有效的平衡.

- 利用时间局部性优化更新操作.在更新操作中,由于时间局部性,最近更新的边更可能被频繁访问.为此,HGraph在执行更新操作时,会先从边块中删除旧数据,并将更新后的数据作为新边插入到边块尾部.通过这种方式,更新后的边更靠近边块尾部,从而有效降低了平均访问成本,提高了系统的访问效率.

- 部分有序策略提升边查找效率.在事务型工作负载中,除插入操作外,其余操作均需定位边的具体位置,因此无法避免地涉及扫描操作.为减少搜索成本,维护有序数据是一种常见策略.然而,完全有序的设计会显著降低系统性能,特别是在插入频繁的场景.此外,由于HGraph采用属性图模型,边的属性大小是不确定的,因此在查找边的属性位置时,必须依靠扫描,无法通过二分查找优化.为此,HGraph引入了部分有序策略,即在保证边块之间存在有序关系的同时,允许每个边块内部保持无序.与原先需要扫描整个邻接链表的方案相比,该策略使每次查找操作的扫描范围限制在单个边块内,同时保留了追加写的插入方式,在查找效率与插入性能之间实现了良好的平衡.

- 适应度数分布的分层数据结构.在真实世界的图数据中,顶点的度数分布通常呈幂律分布.为优化低度顶点的内存利用效率,HGraph采用了小块分配策略,初始边块的大小仅足够存储2、3条边,且与低度顶点相关的操作开销较低,因此无需引入前述优化策略.当边块填满后,系统会将所有数据复制到一个新的边块中,且新边块的大小为原边块的两倍.随着边块大小逐步增大,当达到设定的阈值(例如4KB)时,HGraph会在边块中引入布隆过滤器,以加速边的搜索过程.同时,当边块填满后,系统不再继续扩大边块的大小,而是分配一个新的相同大小的边块,并将部分边迁移至新边块中.具体而言,HGraph会首先扫描边块中的所有边,计算顶点ID的中位数,并依据该中位数将边分配到两个边块中,从而建立块间的顺序关系.这种设计能够适应图数据的度数分布,可以优化存储资源的利用.

4 事务处理流程

本节讨论HGraph的事务执行机制,其主要创新体

现在新颖的多版本并发控制实现上.与传统基于版本链的实现方式不同,HGraph将一个顶点关联的所有边的修改集中存储到一个撤销日志中,这种设计不仅节省了空间开销,还通过高效的顺序内存访问完成回滚操作,避免了高成本的指针追逐操作.此外,在事务处理流程的设计中,HGraph根据底层数据结构的特性,对遍历操作中的回滚过程进行了优化,减少了不必要的检查与指针追逐操作,提高了版本控制的效率.并且HGraph引入了乐观并发控制与写时复制策略,增强了系统的并发处理能力.

- 时间戳机制.HGraph通过维护两个全局时间戳来管理事务执行:全局提交ID和全局事务ID,分别初始化为0和1.每个事务在创建时会被赋予一个起始时间戳,该时间戳被初始化为全局提交ID,用于标识该事务可见的数据版本.此外,写事务还会获得一个事务ID,其值被初始化为当前的全局事务ID,并在创建后将全局事务ID的值加1.在事务提交阶段,写事务会获取一个提交时间戳:系统将全局事务ID自增1,并将自增后的值作为该写事务的提交时间戳.随后,写事务使用该时间戳更新全局提交ID,以表明该更新已对后续事务可见.

- 写事务的执行流程.事务ID为TID的写事务的执行流程,主要包括以下几个步骤:1)数据锁定:在顶点粒度上锁定相关数据,以确保不存在写写冲突.2)旧数据读取:读取目标数据,若数据不存在,则返回空.3)日志条目构建:根据读取的旧数据生成相应的日志条目,其中提交时间戳初始设置为-TID,“-”表示事务尚未提交.4)撤销日志更新:将构建的日志条目及其对应的旧版本数据追加写到源顶点对应的撤销日志中.旧版本数据不仅作为备份以支持事务回滚,同时还作为历史数据版本供长时间运行的事务读取.5)底层数据修改:对于插入操作,HGraph将边单元及其属性追加写到之前查询到的边块中,若边块已满,则可能触发扩展或分裂;对于更新和删除操作,则直接对边进行标记.若边块中的标记数量达到特定阈值,则触发数据迁移以消除内存碎片.6)事务提交或回滚:在提交阶段,事务首先获取当前全局事务ID,计算提交时间戳,然后用其替换日志条目中的原有提交时间戳(即-TID),并更新全局提交ID,最后释放锁定的资源.若事务中止,则需根据撤销日志中的条目进行回滚操作,恢复数据至初始状态.

- 单条边读取的执行流程.在HGraph中,基础的

读操作无需获取锁. 对于一个开始时间戳为 RID 的事务, 读取单条边的流程如下: (1) 边块定位与扫描: 事务首先定位可能存储目标边的边块, 扫描边块以找到目标边, 并复制该边的属性数据. (2) 撤销日志检查: 事务从源顶点的撤销日志末端开始反向扫描 (若日志为空则跳过), 并根据目标边的目的顶点 ID 和标签 ID, 检查相关的日志条目. (3) 版本回滚: 当日志条目的时间戳 ts 满足 $ts \leq 0 \cup ts > RID$ 时, 事务从撤销日志中获取旧版本数据, 并用其覆盖复制的边属性, 从而实现版本回滚. (4) 提前退出优化: 由于同一顶点上的写事务的提交时间戳是递增的, 事务可以利用这一特性, 在撤销日志扫描过程中提前退出. 当事务发现某个日志条目的时间戳 ts 满足 $0 < ts \leq RID$ 时, 扫描可立即停止, 以避免不必要的数据访问.

最终, 事务获得该边的可见版本. 例如, 在图 1 中, 一个开始时间戳为 T_u (假设 $T_u > T_z$) 的事务需要读取标签 ID 为 0 的边 (0, 1) 的属性. 事务首先需要从边块中获取边的属性, 在该例中为“Jimmy”, 然后扫描顶点 0 的撤销日志. 基于日志条目 (1, 0, $-T_z$, 4), 事务将复制的边属性修改为“Gary”. 由于下一个日志条目满足提前退出条件, 撤销日志的扫描提前结束, 最终确定该边属性的可见版本为“Gary”.

- 邻边扫描的执行流程. 扫描顶点的邻接链表包括以下几个步骤: 首先, 事务根据源顶点 ID 和标签 ID 定位到第一个边块的地址. 在读取边块数据时, 事务会根据撤销日志对所有更新进行撤销, 以获取可见版本. 针对这一过程, HGraph 进行了优化: 在从尾到头的边块扫描过程中, 如果找到一条无需撤销操作的边, 则可以推断边块中后续扫描的边也无需进行撤销操作, 从而减少不必要的检查. 这一优化的依据在于, 根据 HGraph 的数据结构设计, 所有需要撤销操作的边 (如新插入或更新的边) 总是位于边块的尾部. 完成边块中所有边的读取后, 事务继续处理被删除的边, 将已被删除但仍应对当前事务可见的边数据恢复. 随后, 事务获取指向下一个边块的指针, 重复上述步骤, 直到获得的指针为空.

- 乐观并发控制与写时复制策略. 为了应对读操作期间边块修改可能引发的冲突, HGraph 引入了乐观并发控制策略. 具体而言, 每个边块均维护一个计数器, 初始值为 1. 当写事务执行可能导致冲突的写操作时 (例如由于边块写满引发的扩展或分裂, 或删除边的数量达到阈值后引发的数据迁移), 该计数器会被暂时置

为 0. 写操作完成后, 计数器的值会更新为原值加 1. 读事务在访问边块之前, 会首先检查计数器的值. 如果计数器为 0, 表示该边块正在被修改, 读事务将等待计数器恢复为非 0 后再继续读取. 读操作完成后, 读事务会验证计数器的值是否发生变化, 若计数器更新, 则读事务需重做读取流程.

此外, 由于扫描边块的时间相对较长, 若写事务在执行可能引起冲突的写操作时, 发现当前边块正在被其他事务扫描, 则会采用写时复制策略. 该策略通过分配一个新的边块, 复制原始数据, 应用更新, 并替换原始边块, 以避免冲突. 原始边块将在所有扫描事务完成后被垃圾回收, 从而确保资源的高效利用.

5 实验

本节首先介绍实验的具体设置, 然后展示实验所得的结果, 并对其进行详细分析和讨论.

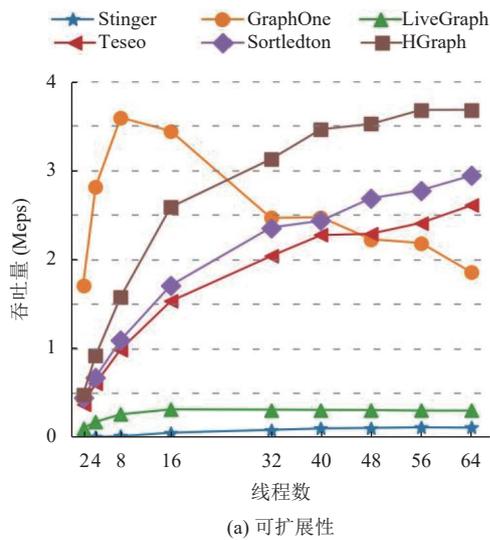
5.1 实验设置

- 数据集. 为了验证方法的有效性, 本研究参考现有工作, 选取了 Dota-league、Graph500 和 Uniform 这 3 类数据集进行实验评估, 数据集的具体信息如表 1 所示. 其中 Dota-league (对应图 2 中的 Dota) 是一个来自实际应用场景的图数据集, 其顶点度数分布呈现轻微的指数趋势. Graph500-x (对应图 2 中的 G500-x) 是一种基于幂律分布的合成图, 其中缩放因子 x 用于表示图的规模, 每当 x 增加 1, 图中的顶点和边的数量均会翻倍^[30]. 为进一步验证方法的适用性, 本研究还引入了 Uniform-x 数据集 (对应图 2 中的 Uni-x), 该数据集的大小与 Graph500-x 相同, 但其顶点的度数分布为均匀分布.

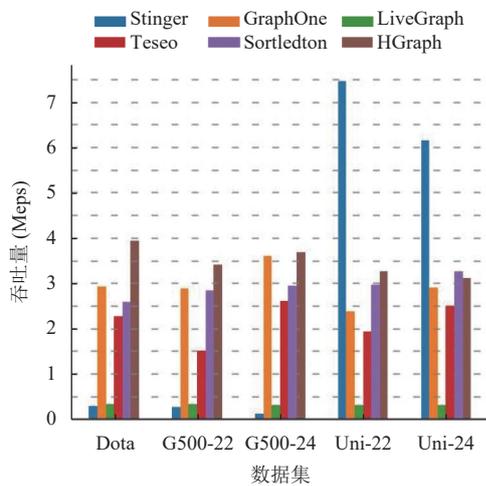
表 1 实验所用数据集介绍

图数据集	顶点数	边数
Dota-league	$\approx 61 \times 10^3$	$\approx 51 \times 10^6$
Graph500-22	$\approx 2.4 \times 10^6$	$\approx 64 \times 10^6$
Uniform-22	$\approx 2.4 \times 10^6$	$\approx 64 \times 10^6$
Graph500-24	$\approx 8.8 \times 10^6$	$\approx 260 \times 10^6$
Uniform-24	$\approx 8.8 \times 10^6$	$\approx 260 \times 10^6$

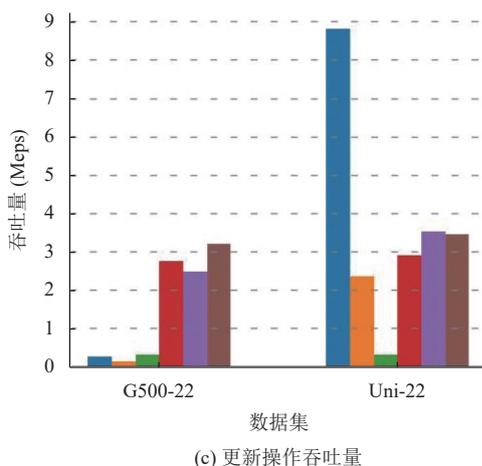
- 对比系统. 本研究选取了多个先进的图存储系统作为比较对象, 包括: Stinger、GraphOne、LiveGraph、Teseo 和 Sortledton. 其中, GraphOne 和 LiveGraph 支持可选的基于磁盘的存储模式, 而其余系统均为纯内存图存储系统. 为了保证对比实验的公平性, 本研究在所有实验中统一禁用了各系统的磁盘日志功能.



(a) 可扩展性



(b) 插入操作吞吐量



(c) 更新操作吞吐量

图 2 插入性能和更新性能测试实验结果

● 比较方法. 本文的实验评估主要依赖于 LDBC Graphalytics Benchmark^[34], 该基准测试包含 6 种算法, 包括广度优先搜索 (breadth first search, BFS)、单源加

权最短路径 (single source shortest path, SSSP)、弱连接分量 (weakly connected components, WCC)、PageRank、局部聚类系数 (local clustering coefficient, LCC) 和基于标签传播的社区检测 (community detection via label propagation, CDLP) 算法. 上述算法的输出均为确定性结果, 实验过程中对其输出值进行了验证. 此外, 由于 Graphalytics 仅包含分析型工作负载, 本文使用以下两种方式来测试系统在事务型工作负载上的性能: (1) 以随机顺序插入图的所有顶点和边. (2) 通过插入和删除临时边来模拟图的动态更新操作, 其中临时边的数量分布与顶点的度数分布一致. 为进一步评估系统在混合工作负载下的性能, 本研究通过在执行更新操作的同时运行 PageRank 算法来模拟混合工作负载场景, 以测试系统在该场景下的表现.

● 实验环境. 实验均在一台配备双插槽 Intel Xeon Gold 5218R 处理器的服务器上进行, 每个处理器有 20 个物理核心, 整机内存容量为 192 GB. 所有系统均使用 GCC v10.2 编译, 编译选项设置为-O3. 每组实验重复运行 5 次, 取其平均值进行报告.

5.2 实验结果

5.2.1 插入性能分析

本实验测试了各系统在单条边插入场景下的吞吐量表现. 在实验过程中, 图数据集的所有边按随机顺序逐条插入. 在插入每条边时, 首先检查其关联的两个顶点是否已存在, 若不存在则插入顶点; 随后检查该边是否已存在, 若不存在则继续执行插入操作, 否则返回错误. 为保证操作的原子性和隔离性, LiveGraph、Sortledton、Teseo 以及 HGraph 将边插入的所有步骤封装为一个事务. 而其他系统无法保证该操作的原子性和隔离性, 其中 GraphOne 更是无法检查目标边是否已存在.

本研究首先测试了各系统的可扩展性, 以确定最佳写线程数量. 图 2(a) 展示了所有系统在 Graph500-24 数据集上, 线程数从 2 增加到 64 时的插入操作吞吐量 (单位: 每秒百万条边, million edges per second, Meps). 在图 2(a) 的横坐标设置中, 本文在并发线程数较小时采用对数增长策略, 以高效观察系统性能随并发程度提升的整体变化趋势; 在并发线程数较大时, 则采用线性增长的方式, 进一步观察系统在高并发时的性能表现, 从而识别潜在的扩展性瓶颈. 图 2(b) 则报告了各系统在不同数据集上使用最佳线程数时的吞吐量表现.

从图 2(a) 可以看出, HGraph、Teseo 和 Sortledton 均表现出良好的可扩展性, 能够扩展至 64 个线程; 相比之下, GraphOne 的插入性能在线程数超过 8 时开始下降, 主要原因是其写缓冲区存在严重的资源争用问题. LiveGraph 的扩展能力则在 16 个线程时达到瓶颈, 这主要归因于其采用的顶点粒度锁机制. 在高负载下, 该机制易成为系统性能的限制因素. 当线程数超过 16 时, 数据插入过程中的事务冲突显著增加, 从而导致插入性能的下降.

对于幂律图, 从图 2(b) 中可以看出, HGraph 的性能表现最佳. 这主要得益于其采用的追加写方法, 以及通过布隆过滤器高效检测边是否已存在的策略. 在绝大多数情况下, HGraph 的插入操作能够以常数时间完成, 从而显著提升了插入操作的吞吐量. 例如, 与 Sortledton 相比, 其插入操作吞吐量提升了 14.9% 到 41.2%. GraphOne 的表现次之, 这是因为其不检测边的存在性, 且通过循环缓冲区批量插入数据, 充分利用所有线程并行地将插入操作应用于主数据结构^[30]. Teseo 和 Sortledton 尽管可以较为高效的检查边的存在性, 但其插入操作容易引起数据迁移, 尤其是 Sortledton. Sortledton 为保持全局有序, 通常会引起频繁的数据迁移, 从而导致插入性能降低. LiveGraph 同样采用了追加写方法并结合布隆过滤器进行边的存在性检测. 然而, 由于其缺乏分块策略, 当布隆过滤器判断边可能存在时, 会产生较高的搜索成本. 此外, 其性能瓶颈可能更多地来源于事务管理器的实现细节, 而非设计层面的因素.

对于均匀图, 从图 2(b) 中可以看出, Stinger 的性能表现最佳. 其在幂律图与均匀图上的显著性能差异, 主要归因于其通过线性搜索检查边的存在性. 在图的度数呈幂律分布时, 这一操作的开销显著增加, 从而导致性能下降. 该结果表明, 对于均匀图, 采用线性搜索来判断边的存在性具有更高的效率.

5.2.2 更新性能分析

本研究测试了各系统在执行边插入和删除混合操作时的性能表现, 实验设置与 Teseo 的方法保持一致. 实验分为两个阶段: 首先, 前 10% 的操作用于加载完整的图数据集; 随后, 在剩余 90% 的操作中, 通过混合执行临时边的插入和删除操作, 并维持图规模的稳定, 从而模拟实际更新应用场景, 最终计算平均吞吐量. 图 2(c) 展示了在可扩展性实验中确定的最佳写线程数量下, 各系统在 Graph500-22 和 Uniform-22 数据集上的平均吞吐量表现.

在 Graph500-22 数据集上, HGraph 展现了更优的性能, 而在 Uniform-22 数据集上, 其表现略逊于 Sortledton. 与实验一相比, HGraph 的吞吐量有所下降, 主要原因在于删除操作引起的数据迁移开销. 但由于 HGraph 采用了分块存储策略和懒惰处理策略, 数据迁移对性能的影响有所缓解, 从而使得性能下降幅度较小. 同样, Sortledton 的性能也略有下降, 但由于其无需进行数据迁移操作, 只需要在执行删除操作时维护版本链, 因此性能下降较少. 然而, 这种设计使被删除的边未从数据结构中移除, 会影响后续的数据访问. 而 Teseo 会从删除操作中获益, 因为删除操作释放了其使用的胖树结构中的存储空间, 从而降低了再平衡和叶节点分裂的频率. LiveGraph 和 Stinger 将删除操作视为新边的插入, 因此其性能表现与实验一相似. GraphOne 的性能则显著降低, 主要原因是其在定位待删除边时面临较大的性能开销.

5.2.3 分析负载性能分析

本实验通过在各系统中执行相同的算法来测试其处理分析负载时的性能, 并且选择 Dota-league、Graph500-22 和 Uniform-22 作为数据集的代表. 对于 BFS、Page-Rank、SSSP 和 WCC 算法, 本研究采用 GAP BS (graph algorithm platform benchmark suite) 提供的标准实现版本^[35]; 对于 LCC 和 CDLP 算法, 本研究基于 Graphalytics 规范进行了自定义并行实现. 实验以 CSR (静态图的最佳通用基准) 作为标准, 对各系统的运行时间进行归一化处理. 图 3 展示了不同系统的减速比, 减速比是系统运行时间与 CSR 的运行时间的比值, 算法名称下标注了此算法在 CSR 上的运行时间, 作为基准值.

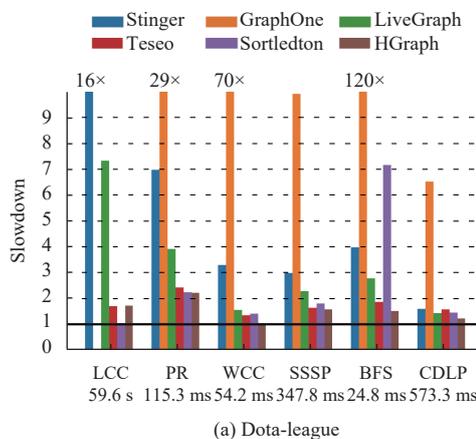


图 3 分析负载性能测试实验结果

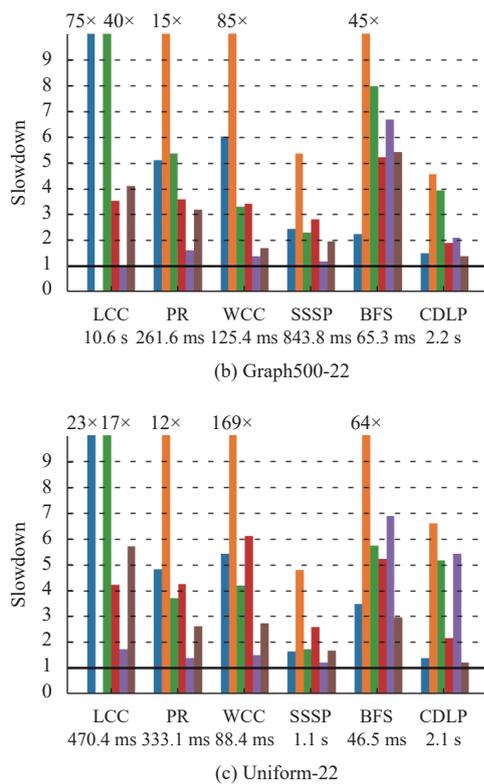


图3 分析负载性能测试实验结果(续)

图3中超出显示范围的数值,标注了相对于基准值的倍数,表示对应系统无法在1h内完成计算任务。可以看出,HGraph、Teseo和Sortledton均展现了较好的性能。其中,HGraph的性能优势主要得益于其数据结构设计,该结构保证遍历操作主要通过高效的顺序内存访问完成。Teseo采用类CSR的数据结构,这种结构同样能够高效支持遍历操作。Sortledton针对交集操作进行了优化,其使用的有序数据结构虽然会略微降低更新操作的吞吐量,但在复杂的图模式匹配任务中展现出了非常优异的性能。

HGraph在3个图数据集上的性能均优于Teseo。然而,与Sortledton相比,HGraph仅在真实图数据集上表现出优势,而在两个合成图数据集上仅在某些算法中表现更优,其余算法的性能稍差。这是由于在本实验中,Sortledton使用了其专有的算法接口,从而避免了迭代器相关的开销。此外,Sortledton在LCC算法中引入了归并排序优化,其实现方式与其他系统不同,能够充分利用Sortledton全局有序的特性,因此在性能上甚至超过了CSR。此外,Sortledton和Teseo使用加权图模型,其边属性大小是固定的,相比于HGraph,无需额外空间来维护边属性大小。这使得在遍历相同数量的

边时,它们访问的数据量较少,且无需管理边属性的位置,因而理论上应具有更高的性能。然而,HGraph的性能与Sortledton相比下降幅度较小。这是因为Sortledton为维持全局有序并减少插入操作引起的数据迁移开销,其边块大小只能设置较小。因此Sortledton在遍历过程中引入了较多的指针追逐操作,导致其在平均顶点度数较高的Dota-league数据集上执行效率较低。

相较之下,GraphOne的性能较差,主要原因在于其需要将每个顶点的邻居复制到用户提供的向量中以供访问。Stinger的边块最多仅能存储14条边,因而引入了大量指针追逐操作,导致性能较低。而LiveGraph虽然在遍历过程中避免了指针追逐操作,但其多版本并发控制机制为边的访问引入了额外开销,从而影响了整体性能。

5.2.4 混合负载性能分析

本实验通过并发执行PageRank算法和更新操作,以模拟混合工作负载场景。该实验仅适用于支持事务处理的系统,包括LiveGraph、Sortledton和HGraph。表2展示了在8-32个分析线程与32个写线程组合下,PageRank算法的延迟和更新操作的吞吐量。

表2 混合负载性能测试实验结果

线程数	Sortledton		LiveGraph		HGraph	
	延迟(s)	吞吐	延迟(s)	吞吐	延迟(s)	吞吐
8	6.72	3.28	6.86	0.31	7.22	3.31
16	3.48	3.18	4.13	0.32	3.70	3.26
32	2.70	3.09	2.90	0.31	2.61	3.18

将混合负载与独立运行更新操作进行对比分析,结果显示,HGraph的吞吐量最大下降了8%,Sortledton下降了12%,而LiveGraph几乎未受影响。HGraph的吞吐量下降主要是因为其采用了写时复制策略,尽管该策略能够保证读事务的扫描操作性能,但却引入了写放大问题,从而影响了写事务的执行效率。Sortledton的性能下降则是由于其读事务需要以顶点为粒度加读锁,这在一定程度上阻碍了写事务的执行。随着分析线程数量的增加,这两个问题更加突出,导致HGraph和Sortledton的吞吐量随分析线程数量的增加而下降。相比之下,LiveGraph采用日志模型的数据结构,能够在不受读事务影响的情况下顺利执行写事务,从而使其性能保持稳定。

在PageRank算法执行过程中,HGraph新引入的MVCC实现相较于传统版本链减少了指针追逐操作和

不必要的检查,因此,当分析线程数达到 32 时, HGraph 的性能超过了 Sortledton 和 LiveGraph. 尽管 Sortledton 使用的是加权图模型,使其在执行该算法时访问的数据量较少,理论上应具有更优的性能,但由于其在数据回滚过程中引入了大量指针追逐操作,实际性能表现低于 HGraph. 而 LiveGraph 因其数据结构中保留了边的所有历史版本,导致在算法执行过程中访问了大量无关数据,从而降低了其性能表现.

6 总结和展望

综上所述,本研究提出了一种适用于混合工作负载的图数据结构,能够高效地处理图上的事务型和分析型工作负载. 此外,本研究还设计了一种基于撤销日志的新型多版本并发控制实现,该实现不仅内存高效,还通过减少不必要的检查和内存追逐操作,提高了扫描操作的性能. 该方法具有较好的通用性,可以广泛应用于键值数据库和关系型数据库中. 进一步地,本研究对事务处理流程进行了优化,提升了系统的并发能力. 未来的工作计划包括将当前基于顶点粒度的锁机制替换为更细粒度的锁或无锁同步机制,以进一步提高系统的并发性能. 同时,还将探索更加高效的事务处理流程,以进一步提升系统的整体吞吐量和响应速度.

参考文献

- 李俊逸, 王卓, 马鹏玮. 图数据库技术发展趋势研究. 信息通信技术与政策, 2021, 47(5): 67–72. [doi: [10.12267/j.issn.2096-5931.2021.05.013](https://doi.org/10.12267/j.issn.2096-5931.2021.05.013)]
- 王宁, 张伟, 王佳慧, 等. 一种关系-图数据库混合存储系统设计. 北京信息科技大学学报, 2022, 37(1): 58–64, 70.
- Li CJ, Chen HZ, Zhang S, *et al.* ByteGraph: A high-performance distributed graph database in ByteDance. Proceedings of the VLDB Endowment, 2022, 15(12): 3306–3318.
- Chen HZ, Li CJ, Zheng CG, *et al.* G-tran: A high performance distributed graph database with a decentralized architecture. Proceedings of the VLDB Endowment, 2022, 15(11): 2545–2558.
- Zhu XW, Chen WG, Zheng WM, *et al.* Gemini: A computation-centric distributed graph processing system. Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016. 301–316.
- Wang JY, Li TL, Song HZ, *et al.* PolarDB-IMCI: A cloud-native HTAP database system at Alibaba. Proceedings of the ACM on Management of Data, 2023, 1(2): 199.
- Vera-Baquero A, Colomo-Palacios R, Molloy O. Real-time business activity monitoring and analysis of process performance on big-data domains. Telematics and Informatics, 2016, 33(3): 793–807.
- Cao SS, Yang XX, Chen C, *et al.* TitAnt: Online real-time transaction fraud detection in ant financial. Proceedings of the VLDB Endowment, 2019, 12(12): 2082–2093.
- Neo4j. <https://neo4j.com>. [2024-11-09].
- Fan WF, He T, Lai LB, *et al.* GraphScope: A unified engine for big graph processing. Proceedings of the VLDB Endowment, 2021, 14(12): 2879–2892.
- Shen SJ, Yao ZH, Shi L, *et al.* Bridging the gap between relational OLTP and graph-based OLAP. Proceedings of the 2023 USENIX Annual Technical Conference. Boston: USENIX Association, 2023. 181–196.
- Wen S, Fokoue A, Srinivas K, *et al.* SQLGraph: An efficient relational-based property graph store. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. Melbourne: ACM, 2015. 1887–1901.
- Kumar P, Huang HH. GraphOne: A data store for real-time analytics on evolving graphs. ACM Transactions on Storage (TOS), 2019, 15(4): 29.
- Ediger D, McColl R, Riedy J, *et al.* STINGER: High performance data structure for streaming graphs. Proceedings of the 2012 IEEE Conference on High Performance Extreme Computing. Waltham: IEEE, 2012. 1–5.
- Macko P, Marathe VJ, Margo DW, *et al.* LLAMA: Efficient graph analytics using large multiversioned arrays. Proceedings of the 31st IEEE International Conference on Data Engineering. Seoul: IEEE, 2015. 363–374.
- Ren K, Zheng Q, Arulraj J, *et al.* SlimDB: A space-efficient key-value storage engine for semi-sorted data. Proceedings of the VLDB Endowment, 2017, 10(13): 2037–2048.
- Dubey A, Hill GD, Escrivá R, *et al.* Weaver: A high-performance, transactional graph database based on refinable timestamps. Proceedings of the VLDB Endowment, 2016, 9(11): 852–863.
- NebulaGraph. <https://www.nebula-graph.io/>. [2024-11-09].
- Pandey P, Wheatman B, Xu H, *et al.* Terrace: A hierarchical graph container for skewed dynamic graphs. Proceedings of the 2021 International Conference on Management of Data. ACM, 2021. 1372–1385.
- Yang K, Ma XS, Thirumuruganathan S, *et al.* Random walks on huge graphs at cache efficiency. Proceedings of the 28th

- ACM SIGOPS Symposium on Operating Systems Principles. ACM, 2021. 311–326.
- 21 Wheatman B, Xu H. Packed compressed sparse row: A dynamic graph representation. Proceedings of the 2018 IEEE High Performance Extreme Computing Conference. Waltham: IEEE, 2018. 1–7.
- 22 Bender MA, Hu HD. An adaptive packed-memory array. Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. Chicago: ACM, 2006. 20–29.
- 23 Zhu XW, Feng GY, Serafini M, *et al.* LiveGraph: A transactional graph storage system with purely sequential adjacency list scans. Proceedings of the VLDB Endowment, 2020, 13(7): 1020–1034.
- 24 Shi JF, Wang B, Xu Y. Spruce: A fast yet space-saving structure for dynamic graph storage. Proceedings of the ACM on Management of Data, 2024, 2(1): 27.
- 25 Han WT, Miao YS, Li KW, *et al.* Chronos: A graph engine for temporal graph analysis. Proceedings of the 9th European Conference on Computer Systems. Amsterdam: ACM, 2014. 1.
- 26 Cheng R, Hong J, Kyrola A, *et al.* Kineograph: Taking the pulse of a fast-changing and connected world. Proceedings of the 7th ACM European Conference on Computer Systems. Bern: ACM, 2012. 85–98.
- 27 Cheng YL, Ma Y, Jiang H, *et al.* TgStore: An efficient storage system for large time-evolving graphs. IEEE Transactions on Big Data, 2024, 10(2): 158–173.
- 28 Prabhakaran V, Wu M, Weng XT, *et al.* Managing large graphs on multi-cores with graph awareness. Proceedings of the 2012 USENIX Annual Technical Conference. Boston: USENIX Association, 2012. 41–52.
- 29 Neumann T, Mühlbauer T, Kemper A. Fast serializable multi-version concurrency control for main-memory database systems. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. Melbourne: ACM, 2015. 677–689.
- 30 Fuchs P, Margan D, Giceva J. Sortedton: A universal, transactional graph data structure. Proceedings of the VLDB Endowment, 2022, 15(6): 1173–1186.
- 31 De Leo D, Boncz P. Teseo and the analysis of structural dynamic graphs. Proceedings of the VLDB Endowment, 2021, 14(6): 1053–1066.
- 32 Kemper A, Neumann T, Finis J, *et al.* Transaction processing in the hybrid OLTP&OLAP main-memory database system hyper. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering. 41–47.
- 33 Chen HZ, Li CJ, Fang JC, *et al.* Grasper: A high performance distributed system for OLAP on property graphs. Proceedings of the 2019 ACM Symposium on Cloud Computing. Santa Cruz: ACM, 2019. 87–100.
- 34 LDBC graphalytics benchmark specification. https://github.com/ldbc/ldbc_graphalytics_docs. [2024-11-09].
- 35 Reference implementation of the GAP benchmark suite. <https://github.com/sbeamer/gapbs>. [2024-11-09].

(校对责编: 张重毅)