计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(10):65−74 [doi: 10.15888/j.cnki.csa.009274]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于持久性内存的属性图存储系统①

卢明祥, 吕　敏

(中国科学技术大学 计算机科学与技术学院 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: 属性图是一种流行的图数据模型, 在各种图系统中得到了广泛应用. 然而, 面向事务型负载的图数据库系统在执行图分析任务的场景下面临着高延迟等挑战. 传统的图分析系统往往是基于简单图模型, 而且大多不支持图的事务型负载. 因此, 迫切需要一个能够在属性图上高效处理事务型负载和图分析任务的图存储系统. 持久性内存的问世, 使得我们有机会重新设计图存储系统, 以充分发挥这种设备的特点. 为此, 本文提出了一种基于持久性内存的属性图存储系统, 名为 TAG. TAG 采用了一种新颖的混合架构的图存储方式, 以充分发挥持久性内存和主存的优势. 其次, 通过拓扑和索引结合的方式, 将图的拓扑嵌入到系统的索引中以加速图的拓扑查询. 最后, TAG 通过基于标签的方式来组织图的属性数据, 进一步优化图的属性访问. 实验结果表明, TAG 显著优于其他图数据库系统, 与图分析系统相比, TAG 也有着相近的性能表现.

关键词: 图存储; 属性图; 索引; 持久性内存; 混合事务分析处理

引用格式: 卢明祥,吕敏.基于持久性内存的属性图存储系统.计算机系统应用,2023,32(10):65–74. http://www.c-s-a.org.cn/1003-3254/9274.html

Property Graph Storage System Based on Persistence Memory

LU Ming-Xiang, LYU Min

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

Abstract: A property graph is a popular graph data model that has been widely used in various graph systems. However, when coming to graph analysis workloads, graph database systems for transactional workloads encounter challenges in terms of high latency. Traditional graph analysis systems are geared towards simple graph models and have limited transactional workload support of graphs. Therefore, there is a growing demand for a graph storage system that can efficiently handle both graph analysis tasks and transactional workloads on property graphs. The emergence of persistent memory provides people with an opportunity to redesign graph storage systems to fully leverage the advantages of this device. To this end, this study proposes TAG, a persistent memory-based property graph storage system. TAG adopts a novel hybrid architecture for graph storage to fully utilize the advantages of persistent memory and main memory. Secondly, by combining topology and index into one, TAG embeds the graph topology into the system index to accelerate queries on the graph topology. Finally, by organizing the graph's property data based on labels, TAG further optimizes access to graph properties. Experimental results show that TAG is significantly better than other graph database systems and has comparable performance to graph analysis systems.

Key words: graph storage; property graph; index; persistent memory; hybrid transaction/analytical processing (HTAP)

① 基金项目: 国家自然科学基金面上项目 (62172382)
收稿时间: 2023-03-03; 修改时间: 2023-05-11; 采用时间: 2023-05-23; csa 在线出版时间: 2023-08-21
CNKI 网络首发时间: 2023-08-22

1 引言

图 (graph) 是一种经典的数据结构, 因为它能够有效地表达数据之间的关联关系, 所以被广泛应用于各个领域, 如社交网络、道路交通、生物医学和金融等领域. 无论在学术界还是在工业界, 图的相关研究都扮演着越来越重要的角色[1]. 根据 DB-Engines 的数据, 自 2013 年以来, 图数据库一直是增长最快的数据库类型[2], 知名的研究机构 Gartner 也将图数据库列为 2021 年数据和分析领域的 10 大重点趋势之一[3].

近几十年来, 学术界和工业界设计并开发了大量的图系统. 根据功能实现的不同, 它们可以大致地分为两类: 图数据库和图分析系统. 图数据库专注于事务型的负载, 例如插入一个顶点或一条边, 查询某个顶点的属性等. 而图计算系统则聚焦于分析型的负载, 例如基于全图的迭代性计算. 由于二者的访问模式并不相同, 它们采用了不同存储方案来满足各自的需求. 如果要对数据库中的数据进行分析计算, 则必须通过抽取/转换/加载 (extract, transform, load, ETL) 完成数据转换, 但这会带来巨大开销[4].

自第 1 个图计算系统 Pergel[5] 问世以来, 图计算领域涌现出了许多代表性的工作, 例如 GridGraph[6]、GraphChi[7] 等. 这些工作从优化编程模型、数据划分等角度提高了系统的性能, 但它们只考虑了静态的简单图. 就存储层而言, 这些工作多采用压缩稀疏行 (compressed sparse row, CSR) 格式, 虽然这种格式的存储效率很高, 但修改代价也很大, 因此无法支持动态图. 近年来, 也有一些工作开始尝试支持图的动态更新, 例如 GraphOne[8] 和 LLAMA[9], 但这些系统依然使用简单图模型, 只考虑了图的拓扑结构. 然而, 在现实场景中, 拓扑结构只是图的一部分, 更重要的是属性信息. 属性图模型的应用更为广泛, 而且属性数据通常比拓扑数据占据更多的存储空间. 以 Facebook 的社交图为例, 顶点的属性数据平均大小为 87.6 字节, 而拓扑信息和时间戳等元数据则只有 24 字节. 大多数图数据库使用属性图模型存储各类应用中的数据, 例如 Neo4j[10]、NebulaGraph[11]、Microsoft A1[12]、LiveGraph[4] 和 IBM Db2 Graph[13]. 这些系统专注于图的事务型负载, 并采用了不同的方法来存储图, 例如键值存储或关系表. 虽然这些系统做了许多优化, 但是在应对种类繁多的图操作时, 仍然面临挑战.

使用键值系统存储图时[11,14], 通常会把每个顶点和每条边视作单独的键值对. 这种方法可以降低修改数据的成本, 但是会导致图遍历操作变得非常复杂, 因为需要进行大量的随机读操作[15]. 一些系统采用顶点表的方式存储图[4,8], 把每个顶点的所有的邻居边连续地存储在一起. 这种方式可以更有效地支持拓扑查询, 但是这些系统不支持顶点的删除, 而且查询某条边的时间复杂度取决于该顶点的度数. 关系数据库作为一种经典的存储方式, 也被许多的图数据系统作为底层存储方式[13]. 关系数据库提供的数据模型是一个二维表格, 系统把相同类型的顶点或边存储在一个表中, 这对常规查询很有效, 但是访问图的拓扑时需要执行复杂的表连接操作, 从而降低了系统的性能[15]. 还有一些系统使用双向链表[10,12] 来组织数据, 每个顶点都维护了邻居边的引用. 这种方法的好处是实现了无索引邻接, 即每个顶点都可以通过链表访问邻居边, 再通过邻居边的链表访问相邻顶点, 避免了使用全局索引带来的开销, 但访问链表会引入大量的随机读操作.

为了解决上述问题, 我们需要一个基于属性图的混合事务/分析处理系统 (hybrid transaction/analytical processing, HTAP) 以满足事务型和分析型任务的需求. 这样的系统不仅可以支持实时的分析, 而且可以避免 ETL 的开销. 然而, 实现这样的系统面临着许多挑战.首先, 分析型和事务型这两类负载对系统有着不同的要求, 因此在设计上需要进行权衡和优化. 分析型任务要求系统支持图的全局性查询、迭代计算等并快速返回结果, 一般不会修改图的拓扑结构. 而事务型的负载要求系统能够快速地修改图的拓扑和属性, 例如插入一条边或者更新某个顶点的属性等, 并能够查询图的拓扑结构和属性信息.

其次, 存储系统需要支持图的两种访问模式: 拓扑访问和属性访问. 拓扑访问是指从某个顶点出发遍历它所有的邻居边. 属性访问是指对顶点或边的属性进行各种操作, 例如过滤、排序和分组[16] . 因此, 存储系统需要能够高效地访问图的拓扑结构和属性信息, 并支持对相同标签的顶点或边进行分析等.

最后, 系统需要提供高性能的服务. 持久性内存 (persistent memory, PMem) 具有低延迟与持久性的特点, 可以替代传统磁盘. 已有一些工作在 PMem 上构建键值数据库、索引等[17,18]. 但是, 目前还没有工作利用 PMem 来构建图存储系统. 如何结合动态随机存取存

储器 (dynamic random access memory, DRAM) 和 PMem 各自的优势, 构建一个高效的图存储系统, 这也是一个值得探索的问题.

综合以上分析, 本文设计和实现了一个高效的属性图存储系统, 命名为 TAG (storage system for transactional and analytical processing on property graphs). TAG 采用了混合的图存储架构, 它将图的属性信息存储在 PMem 上, 在 DRAM 上存储索引, 以充分发挥 DRAM 和 PMem 的存储特性. 其次, 通过精心设计索引, 把主索引和图拓扑合二为一: 索引不仅维护主键和记录之间的映射关系, 用于快速查询顶点或边的属性, 同时也是图的拓扑. 因此, 仅需借助索引即可实现图的拓扑操作. 最后, 对于存储在 PMem 上的属性数据, 针对属性图的访问模式和 PMem 的特点, 采用了一种新的策略来组织 PMem 中的数据: 按页划分 PMem 的空间, 并根据顶点和边的标签来分配页. 此外, 通过维护偏移表, TAG 提高了图属性数据的访问效率. 本文的主要贡献点总结如下.

(1) 混合存储架构: 基于最新的持久性内存, 设计了一个混合的图存储系统, 以充分发挥两种介质的特点. 具体来说, 将索引放置在 DRAM 中, 而将数据持久地存储在 PMem 中.

(2) HTAP 图存储系统: TAG 同时支持事务型和分析型这两大类负载.

(3) 索引即拓扑: 通过设计合适的主键, TAG 将图的拓扑结构隐式地存储在索引中, 可以有效地支持图遍历, 同时不会占用额外的存储空间.

(4) 基于标签的页面管理方式: TAG 基于顶点和边的标签, 组织持久性内存中的数据, 从而更加高效地支持属性查询.

2 背景介绍

本节将介绍属性图模型及其各种应用, 重点探讨这些图系统所使用的数据模型的局限性, 并详细讨论持久性内存的特点.

2.1 简单图和属性图

简单图是一种经典的数据模型, 由顶点和边组成, 每个顶点由唯一的编号标识, 称之为顶点标识符 (vertex ID, VID). 每条边由其源顶点和目标顶点的 VID 标识. 顶点和边可以带有简单属性, 例如权重. 与之相比, 属性图在顶点和边上带有更多的信息, 用标签 (label) 来指示它们的类型, 相同标签的数据有着同样的数据模式. 类似于关系数据库中的表, 一张表有确定的若干列. 例如图 1 中, 标签为 Person 的顶点会有“firstName” “lastName”和“age”等属性. 边的情况也是如此, 但在某些情况下, 边可以只用来表示连接关系而不携带信息, 例如图 1 中标签为“hasCreator”的边.

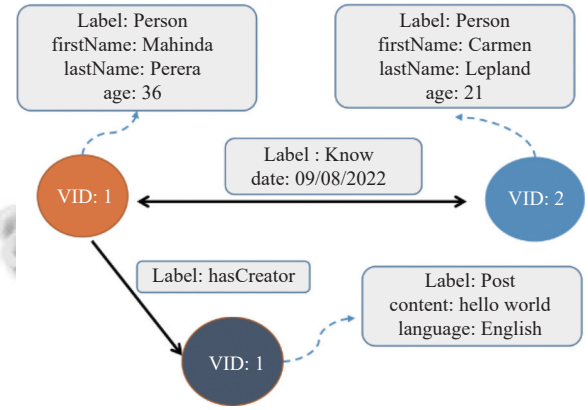

图 1 属性图示例

2.2 图存储系统中的数据模型

在简单图中, CSR 和顶点表 (vertex table) 是最为普遍的两种存储格式. CSR 使用两个数组, 一个顶点数组和一个边数组, 来存储数据. 一个顶点的所有邻居边都连续地存储在边数组中, 而顶点数组中则存储了该顶点的第一条邻居边在边数组中的偏移位置. 大多数图分析系统使用 CSR 的方式存储图数据, 因为它具有理论上最低的存储开销并且支持高效的图查询. 但是, 当需要更新顶点和边时, 需要重构这两个数组. 如图 2 所示, 在示例图中插入一条边时 (即示例图中的<1, 6>), 顶点数组和边数组几乎都需要修改 (图中展示了顶点数组和边数组更新前后的变化), 因此这种巨大的开销使得 CSR 无法用于动态图的应用场景. 在支持动态图的系统中, 顶点表则是一种常见的存储模型. 在顶点表中, 每个顶点的所有邻居边保存在一个数组里, 该数组的地址则保存在顶点数组中. 然而, 顶点的总数会受到顶点数组固定大小的限制, 而查找一条边的时间复杂性取决于该顶点的度数, 这种查询的不稳定性是系统所希望避免的. 此外, 无论是 CSR 还是顶点表, 它们都只考虑图的拓扑结构, 而没有涉及顶点或边的属性的存储.

在支持属性图的存储系统中, 存在多种数据模型, 包括基于日志结构合并树 (log-structured-merge-tree,

LSMT)、关系表和链表等. 这些存储系统能够有效地支持顶点和边的插入以及一些简单的查询, 但对于图分析, 特别是涉及图拓扑的一些复杂查询时, 它们的性能表现无法满足需求. 例如, 有许多图存储系统使用基于 LSMT 的 RocksDB 作为底层存储引擎, 因为它具有出色的写入性能. 然而, LSMT 存在读放大的问题, 特别是涉及范围查询时. 在基于关系表的系统中, 通常会将相同类型的顶点或边存储在单个表中, 这对于常规的查询足够高效的, 然而, 在执行图遍历操作时, 会导致表连接操作, 这种类型的操作开销巨大, 会影响整个系统的性能. 使用链表存储图也是一种常见的选择, 边的插入在链表中非常的简单, 但在查询时, 随机访问带来的开销难以避免.

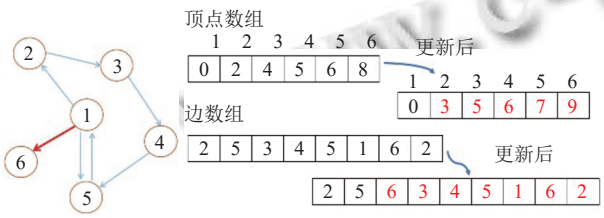

图 2 简单图的 CSR 存储格式及其更新

2.3 持久性内存

随着英特尔的傲腾持久性内存的发布[18,19], 越来越多的人开始关注它的特性以及如何充分利用这种介质. 傲腾持久性内存的存储密度比 DRAM 更高, 它可在内存模式和应用模式下运行. 内存模式下使用傲腾持久性来扩展主内存, 但不具备持久性. 在应用模式下, 傲腾持久性内存可作为持久性存储设备. 在 TAG 中, 使用后者来实现数据持久化. 为了实现更高的并行性, TAG 在交错模式下使用傲腾持久性内存. 在交错模式下, 数据以大小为 4 KB 页面在所有的傲腾持久性内存上交织. 尽管研究发现傲腾持久性内存的特性比较复杂, 但我们对于 DRAM 和傲腾持久性内存有一个基本的认知, 即它们的读写延迟在同一个数量级, 但在随机访问时, 仍然存在几倍的差距.

3 方案设计

本节将介绍一种高效的属性图存储系统—TAG. 该系统采用一种混合架构, 将图的属性和拓扑分别存储在不同的物理存储介质上 (DRAM 和 PMem), 以实现高效访问. 接下来, 我们将首先将介绍系统的架构, 然后详细介绍 TAG 中的两个核心设计.

3.1 系统架构

TAG 系统的总体架构如图 3 所示. TAG 的定位是一个图存储引擎, 最上层是接口层, 以库的形式为各类图应用提供存储服务. 在内部, TAG 由索引层和数据持久层组成. 其中, 系统的主索引同时也是图的拓扑结构, 存储在 DRAM 中以实现高效访问, 而顶点和边的属性存储在 PMem 中. 为了提高读取性能, 系统需要索引来定位 PMem 中数据的位置, 但是维护一个索引会带来额外的存储开销和操作复杂性. 因此, TAG 将图拓扑和索引合二为一, 避免额外的存储开销, 并简化了图拓扑相关的操作. 在接下来的部分中, 我们将详细介绍 TAG 系统的实现.

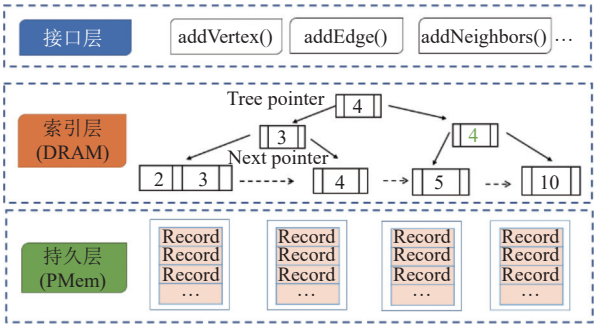

图 3 系统架构

为了支持范围查询, TAG 系统采用 B+树作为主索引. B+树具有良好的查询和更新性能[20], 在工业界特别是在关系数据库中得到了广泛应用, 在学术界也有大量相关的研究[17,20–23].

TAG 的设计需要考虑可扩展性. 传统的 B+树虽然能保证数据一致性, 但需要通过对整棵树加锁来实现, 这会限制写入性能. 无锁的 B+树虽然提供了良好的可扩展性, 但实现过于复杂[23]. 因此, TAG 系统选择采用带有乐观锁耦合 (optimistic lock coupling, OLC) 技术的 B+树作为主索引. 相对于传统 B+树, 带有 OLC 的 B+树在每个节点中添加了一个额外的头部, 用于加锁和版本计数. 为了简便, 在本文中, 我们直接使用 B+树或索引来指代使用了 OLC 技术的 B+树.

3.2 索引即拓扑

本节将讨论 TAG 中用于存储图拓扑和索引信息的方法. 为了高效地访问 PMem 中的数据, TAG 需要使用索引. 虽然一些现有的图存储系统将图拓扑存储

在内存中[16], 并使用顶点表来支持图遍历, 但这种方法会增加额外的内存开销和操作复杂性. 为了避免这种开销, TAG 将图拓扑和索引信息合并在一起, 通过巧妙地设计顶点和边的主键, 将索引本身作为图的拓扑. 这种方法可以有效减少内存开销和操作复杂度.

3.2.1 简单图

为了保证顺序地访问拓扑, 在 TAG 中, 将顶点及其邻居边的主键连续地存储在索引中. 这样, 当执行图遍历等操作时, 只需顺序扫描内存中的索引即可. 接下来, 将详细介绍简单图和属性图中主键的设计.

对于简单图, 每个顶点都有一个唯一的 VID 标识, 每条边由其源顶点的 VID 和目标顶点的 VID 表示. TAG 使用这些 ID 来构建顶点和边的主键. 表 1 展示了图 2 中示例图的部分顶点和边的主键. 如表所示, 每个 VID 由一个 4 比特的二进制数表示. 边的主键由源顶点和目的顶点的 VID 组成. 为了使顶点的主键与边的主键对齐, 顶点主键由 VID (最高 4 位) 和 0 (最低 4 位) 拼接而成. 在这个示例图中, TAG 的主键是一个无符号 8 比特的整数. 在实际应用中, 可以根据图的规模来设置键的位数. 例如, 当主键的长度为 64 位时, 可以编码包含 2^{32} 个顶点和 2^{64} 条边的图.

表 1 简单图中的主键

ID	主键
1	00010000
<1, 2>	00010010
<1, 5>	00010101
<1, 6>	00010110
2	00100000

为了支持高效的图遍历, TAG 需要确保顶点及邻居边的主键连续存储. 这可以利用 B+树的有序性来实现. 以图 1 为例, 对于 VID 为 1 的顶点, 它的主键为 00010000, 后续的主键依次为 00010010、00010101 和 00010110, 分别对应它的邻居边<1, 2>、 <1, 5>、<1, 6>的主键. 当要查询图中 VID 为 1 的顶点的所有邻居时, 只需要在索引中做一次范围扫描, 其扫描的区间是 [00010000, 00100000). 算法 1 详细地描述了该过程. 如第 2 行所示, 根据 ID, 通过位运算构造范围查询的起始键和终止键. 第 3–5 行检查索引的根节点是否为空, 如果为空则说明还未插入数据, 则返回空结果集来处理索引树还未建立的情况. 第 6–9 行在索引建立后, 从根节点出发, 进入对应的孩子节点, 直到找到对应的叶节点. 第 10–14 行定位到起始键. 如果定位失败, 说明用户查询的顶点不存在, 则终止查询并返回空结果集. 如果定位成功, 第 15–18 行顺序扫描叶子节点上的主键, 对满足条件的主键, 通过位操作重新构造 ID 并加入到结果集中. 第 19, 20 行如果当前的叶子节点扫描完毕但范围查询还未结束, 则通过邻居指针跳到邻居节点中, 继续扫描. 第 21, 22 行重复上述循环, 直到扫描的主键大于终止键或者到扫描到了索引的最后一个主键, 扫描结束并返回结果集.

B+树的分支因子 (branch factor) m 决定了节点的容量. 在 TAG 的实现中, m 的默认值被设置为 256, 这也意味着一个节点中的主键个数必须大于 128 并小于 256. 为了最小化随机访问的开销, 主键和相应的指针存储在数组中. 写入顶点或边的过程包括两个步骤: 首先, 数据存储在 PMem 中以实现持久化. 其次, 将相应的主键和指针插入到索引中合适的位置.

算法 1. getNeighbors 伪代码

```
(1) FUNCTION(VID)
(2)   start_key, end_key = constructKeys(VID);
(3)   IF index.root is EMPTY THEN
(4)       RETURN [];
(5)   END IF
(6)   current_node = index.root;
(7)   WHILE not current_node.is_leaf;
(8)       current_node=current_node.get_child( start_key);
(9)   END WHILE
(10)  WHILE current_node is not None:
(11)    FOR key in current_node:
(12)        IF key > end_key:
(13)            RETURN [];
(14)        END IF
(15)        IF key ≥ start_key:
(16)            neighbor_ID = reconstructID(key);
(17)            results.append(neighbor_ID);
(18)        END IF
(19)    END FOR
(20)    current_node = current_node.get_next_leaf();
(21)  END WHILE
(22)  RETURN results;
(23) END FUNCTION
```

3.2.2 属性图

与简单图相比, 属性图中的主键的设计更为复杂, 因为顶点和边都拥有标签. 为了解决这个问题, TAG 采用了标签 ID (label ID, LID) 和顶点/边 ID 的组合来设计主键. 这种设计不仅可以实现对于拓扑的高效访问,

同时还可以隐式地表明顶点和边的标签.

TAG 使用哈希表来维护标签名和 LID 的映射关系, 每个标签都被分配了一个唯一的 LID, 对于图 1 中的示例图有 4 种标签: Person、Post、Know、 hasCreator. 为它们分配的 LID 的二进制表示是 (0001, 0010, 0011 和 0100). 一个图中标签的数量通常是有限的, 而一个标签只需要几个字节的内存来存储它的名称和 LID, 因此哈希表带来的开销很小.

表 2 展示了图 1 中顶点和边的主键设计. 其中, 标签用 4 个比特表示, 最多可以表示 16 种标签. 主键由 16 个比特表示. 具体来说, 对于一个顶点, 最高的 4 位表示它的 LID, 接下来的 4 位表示顶点的 VID, 最低的 8 位设置为 0 以和边的主键对齐. 对于一条边, 最高的 8 位表示源顶点的 LID 和 VID, 后面的 8 位表示边的 LID 和目的顶点的 VID. 这种设计确保一个顶点的主键和它的邻居边在索引中连续存储, 从而可以通过扫描索引实现图的遍历, 并且有利于基于标签的访问模式. 例如, 当需要检索标签为 Person 且 VID 为 1 的所有好友时, TAG 只需要执行范围查询, 其区间为 [0001 000100110000, 0001000101000000). 同时, 通过主键可以直接获取顶点 (边) 的标签.

表 2 属性图中的主键

Label: ID	主键
Person: 1	0001000100000000
Know: (1, 2)	0001000100110010
hasCreator: (1, 1)	0001000101000001
Post: 1	0010000100000000

3.3 基于标签的页设计

由于具有相同标签的顶点或边有相同的结构, 而且许多查询任务只关心特定标签的数据. 因此 TAG 根据标签来组织在 PMem 中数据, 即同一页中只会写入同一种标签的数据, 并使用数组来跟踪这些页面. 通过将索引即拓扑的策略, TAG 可以有效地处理单点读和图遍历操作. 但是, 在需要查询属性数据时, 仍然存在挑战. 在键值存储系统中, 通常将所有属性字段序列化为键值对, 但当只需要查询特定字段时, 这会导致读取放大的问题. 为了解决这个问题, TAG 设计了一个偏移表来跟踪每个字段的位置.

图 4 展示了一个页的结构: 每个页面的开头使用一个字节来加锁, 以确保并发访问的安全性; 接下来是尾指针, 它记录了下一次写入的起始位置; 最后是需要存储的具体数据. 每条记录代表一个顶点或边, 由 3 部分组成: 主键、偏移表和属性. 为了在系统出现故障时重建索引, 主键会再次存储在页中. 偏移表用于跟踪每个可变长度字段的偏移量, 每个字段的长度用一个字节记录. 如果字段的长度小于 255 字节, 则其数据直接存储在偏移表之后. 对于特别大的字段, 例如照片, 则存储在其他位置, 通过维护一个引用来访问. 通过偏移表, TAG 可以直接访问特定字段, 从而减少读取放大的问题.

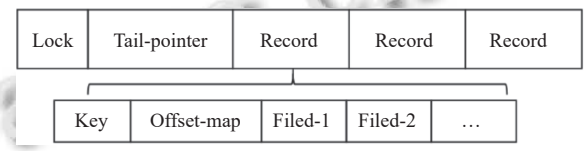

图 4 页的结构

4 实验分析

本节将介绍对 TAG 系统的实验评估, 旨在验证其在处理事务型和分析型任务时的性能优势. 我们在 Linux 系统下实现了图存储引擎 TAG, 采用了本文提出的设计策略, 如 DRAM 和 PMem 的混合架构、索引即拓扑的设计和基于标签的页划分等. 我们将 TAG 与几种主流的图系统进行性能比较, 包括事务型的图数据库和分析型的图计算系统. 我们首先展示了 TAG 与图数据库系统 (Neo4j 和 NebulaGraph) 的比较结果, 由于涉及属性访问, 所以需要使用二级存储介质, 但是 Neo4j 和 NebulaGraph 都没有基于持久性内存的版本, 它们都是基于固态硬盘 (solid-state drive, SSD) 设计和实现的, 因此为了公平地进行比较, 同时考虑到这两个商业数据库不易修改, 我们实现了一个使用 SSD 作为二级存储介质的 TAG 版本. 对于分析型任务, 我们采用 LLAMA 和 GraphOne 作为对比对象, 它们都是内存型的图分析系统. 最后, 我们进行了微基准测试来评估我们设计的各种策略, 以进一步验证它们的有效性.

4.1 实验环境和数据

我们在一台本地的服务器上进行了实验, 该服务器的操作系统是 Ubuntu 20.04, CPU 为 Intel Xeon Gold 5218R, 主存有 192 GB, 傲腾持久性内存有 384 GB. 对于图数据库查询的任务, 我们采用了关联数据基准委员会提供的社交网络基准测试 (linked data benchmark council's social network benchmark, LDBC-SNB), LDBC

提供了多个数据集和典型的查询场景. 此外, 我们还使用了其他一些不同规模的数据集, 具体的描述如表 3. Uniform-500 是一个和 Graph-500 有着相同的规模但是均匀分布的数据集, 即每个顶点的度数都相同, 用于测试数据分布对系统性能的影响. LDBC-SF-1 和 LDBC-SF-3 是同构的属性图, 即标签的类型和数目都相同, 它们之间的唯一区别是数据规模的大小.

表 3 实验数据集

数据集名称	顶点数	边数	类型
Graph-500	4 194 304	67 108 864	简单图
Uniform-500	4 194 304	67 108 864	简单图
Facebook	4 039	88 234	简单图
LDBC-SF-1	1 633 006	13 377 602	属性图
LDBC-SF-3	4 899 018	40 132 806	属性图

4.2 查询延迟

本节将 TAG 与图数据库 Neo4j 和 NebulaGraph 进行比较. 由于 TAG 采用了索引即拓扑的策略, 可以提高对图拓扑的访问性能, 所以我们首先测量了一跳查询延迟, 然后测量了基于 LDBC-SNB 基准测试的平均延迟. 本节报告的所有结果都是 5 次运行的平均值.

基于 TAG 的一跳查询的归一化延迟如图 5 所示, 其中 y 轴采用对数坐标. 为了评估一跳查询的性能, 先后在两个大小不同的数据集上进行实验, 即 Facebook 和 Graph-500. 对于每个数据集, 本文随机选择了 300 个顶点作为一跳查询的源顶点, 然后在 TAG、 Neo4j 和 NebulaGraph 上执行一跳查询. 在较小的 Facebook 数据集中, TAG 的平均延迟为 0.221 ms, 比 Neo4j (10.124 ms) 和 NebulaGraph (11.319 ms) 快 45 到 50 倍. 在更大的 Graph-500 数据集上, TAG 的平均延迟为 30.888 ms, 比 Neo4j (9 196.084 ms) 和 NebulaGraph (752.95 ms) 快 25 到 300 倍.

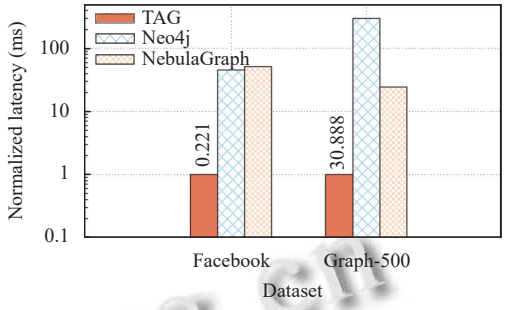

图 5 归一化的一跳查询延迟

TAG 的目标是一个高效的属性图存储引擎, 为了评估 TAG 的性能, 我们使用了著名的 LDBC-SNB 基准测试来评估 TAG 的性能. 实验选取了 4 个典型的负载: 交互式短查询 IS-1 和 IS-3、交互式复杂查询 IC-1 和商业智能查询 BI-1. IS-1 查询单个顶点的属性, IS-3 是从指定顶点出发, 访问它的一跳邻居, 并查询邻居顶点的属性. IC-1 通过指定边的标签来访问多跳邻居顶点的属性, BI-1 则是扫描指定标签的所有顶点并进行分类. 详细描述请参考官方文档[24,25]. 实验在两个不同大小的数据集 (LDBC-SF-1 和 LDBC-SF-3) 上进行. 表 4 中的结果表明, TAG 在所有查询类型中均优于其他数据库, 在面对复杂查询任务时 (IC-1 和 BI-1) 表现突出. 这是由于索引即拓扑的设计提高了图遍历的性能, 通过 DRAM 中的索引可以快速获取顶点和边的标签信息, 从而进行高效的过滤操作. 此外, 具有相同标签的顶点和边在同一页上存储, 进一步提高了基于标签的查询的性能.

表 4 LDBC 查询延迟 (ms)

数据集	图数据库	IS-1	IS-3	IC-1	BI-1
LDBC-SF-1	NebulaGraph	1.164	1.669	212.425	82 819.421
	TAG	0.003	0.015	175.846	274.109
LDBC-SF-3	NebulaGraph	2.889	3.778	580.338	2 873 225.797
	TAG	0.005	0.017	319.545	1 294.813

4.3 分析型负载

TAG 的目标是支持混合负载, 所以 TAG 也和图分析系统 GraphOne 和 LLAMA 进行了比较. GraphOne 是一个图存储系统, 它利用边列表和顶点表的组合进行存储, 边列表作为更新的缓冲区, 顶点表是主要的存储格式. LLAMA 是一个基于多版本的 CSR 格式的图分析系统, 但由于本文的实验数据是一次性写入的, 因此 LLAMA 只生成一个版本. 如前文所讨论的那样, CSR 有着理论最优的性能, 因此我们将其作为实验评估的基准线. 为了评估这些系统在分析型工作负载中的性能, 使用广度优先搜索算法 (breadth first search, BFS) 作为实验中的任务.

如图 6 所示, 使用 CSR 的 LLAMA 有最佳的表现,这是因为在实验设置中, 只生成了一个版本. 然而在实际的动态图场景中, 如果写操作很频繁, LLAMA 会生成多个版本, 这将导致它的效率显著下降, 尽管可以通过归并操作来减少版本数量, 但这也会带来额外的开销. TAG 和 LLAMA 的延迟在同一个数量级, 而与 GraphOne 相比, TAG 在 Uniform-500 和 Graph-500 数据集上分别比 GraphOne 快了 7 倍和 4 倍. 这主要得益于 TAG 只需要顺序扫描索引来访问邻居边, 而 GraphOne 需要访问顶点表和边列表这两个结构. 此外, 只有 TAG 支持属性图, 而 LLAMA 和 GraphOne 都只能处理简单图.

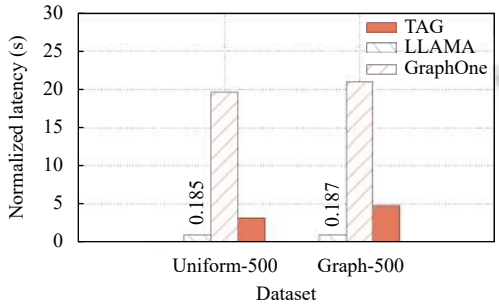

图 6 归一化的 BFS 延迟

4.4 系统设计的效果分析

为了验证 TAG 设计的策略的有效性, 我们通过了一系列的实验来评估 TAG 的两个核心策略: 索引即拓扑和基于标签的页设计.

4.4.1 索引即拓扑的策略

TAG 的一个核心策略是将图拓扑嵌入到索引中, 这样, 图遍历操作可以通过扫描内存中的索引来获取邻居顶点的 ID, 而无需访问二级存储中的数据. 此外, TAG 还通过巧妙的主键设计, 实现了顶点和它的邻居边顺序存储在一起, 保证了顺序读. 基于索引即拓扑的策略, TAG 中提供了两种方式来获取顶点的邻居顶点. 第 1 种方式称为 Index: 它直接扫描主索引, 通过范围查询获取相应的主键, 然后根据主键的构造原则, 通过位操作来构造邻居顶点的 ID. 第 2 种方式称为 Page: 它从内存索引中获取邻居边的地址, 然后访问持久性内存中的记录, 获取邻居顶点的 ID.

为了评估索引即拓扑的策略的性能, 我们在两个大小不同的数据集 Facebook 和 Graph-500 上测试了一跳查询的延迟. 我们随机选择了 300 个顶点作为源顶点, 并分别用 Index 和 Page 两种方式访问它们的邻居. 图 7 展示了两种方式在不同数据集上的延迟情况, 其中纵坐标表示的是归一化的延迟. 可以看出, 在较小的 Facebook 数据集上, Index 方式的延迟为 0.221 ms, Page 方式的延迟为 1.437 ms. 在较大的 Graph-500 数据集上, Index 方式的延迟为 35.888 ms, Page 方式的延迟为 942.231 ms, 两者之间的差距更加明显.

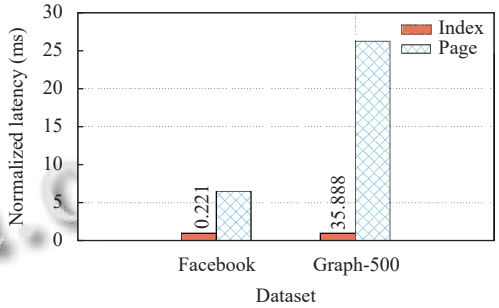

图 7 归一化的一跳查询平均延迟

实验结果验证了我们的预期, Index 和 Page 两种访问方法之间存在显著的性能差异. 这是因为 Index 方式可以避免访问二级存储, 从而大幅降低了延迟. 在我们的实验中, 二级存储设备是持久性内存, 它的读写延迟和内存是一个数量级, 如果二级设备是固态硬盘或者磁盘的话, 这种性能差距会更加巨大. 另外, 访问的边越多, Index 方式的优势就越明显, 因为 Page 方式会导致更多的对二级存储的访问. 同时, Index 方式还保证了对邻居边的扫描是顺序的 (只有跨叶子节点的访问会导致随机访问). 在我们的数据集中, 所有的邻居边都按照顺序插入和存储在二级存储中, 在真实的工作负载中, 数据的插入不会是完全顺序的, 因此访问邻居边会导致大量的对二级存储的随机访问. 相比之下, Index 方式的访问总是顺序的, 因此我们的设计在真实场景中会更加高效.

4.4.2 基于标签的页设计

在基于标签的页面设计中, 我们根据持久性内存的特性和属性图的访问模式, 把具有相同标签的顶点或边组织在一起, 同时通过偏移表来定位每一个属性字段. 为了评估这种策略的效果, 本文在 LDBC-SF-1 数据集上设计了一个查询任务, 用来统计数据集中男性和女性的人数. 该查询涉及读取所有标签为 Person 的顶点, 并读取它们的“gender”属性字段, 来判断读到的数据是男性还是女性, 并统计各自的数量. 实验采用了 3 种方法执行查询任务. 第 1 种是 Get: 它先构造查

询主键, 然后访问索引来获取记录的地址, 再读取所需的字段并计数. 第 2 种是 Index-scan: 它从叶节点开始扫描, 并使用主键的标签位来检查记录的标签, 只有标签为 Person 时, 才读取记录. 最后一种称为 Page-scan: 它通过元数据获取标签为 Person 的数据页面, 并顺序扫描所有记录, 再读取所需的字段.

实验结果验证了我们的预期, 3 种方法之间存在显著的性能差距. 3 种方法的延迟依次为 6.142 ms, 2.023 ms 和 0.718 ms. Index-scan 通过主键隐含的标签信息, 在内存的索引中提前完成了过滤操作. Page-scan 则是避免了索引的开销, 直接扫描相应的页面. 由于 TAG 中数据是基于标签来组织的, 一个页中只存储相同标签的数据, 因此 Page-scan 还实现了对持久性内存的顺序访问.

5 相关工作

在图领域, 相关的工作可以分为两大类: 一类是针对分析工作的图计算系统, 另一类是面向事务型负载的图数据库. 前者包括 Gemini[26] 和 GridGraph[6] 等系统, 它们通过实施图划分算法[27]、 减少通信开销和磁盘 I/O 等不同角度来提高图分析的性能. 这些系统通常采用 CSR 格式进行存储. 在最新的一些工作中, 有些系统在支持分析型负载的同时也能够支持动态图, 例如 LLAMA[9]、Teseo[28] 和 GraphOne[8], 但是这些系统主要专注于分析型负载并且缺乏对属性图的支持.相比之下, TAG 为属性图提供了存储服务, 能够有效地支持分析型和事务型这两类负载.

而由微软等主要科技公司开发的图数据库则基本上是基于属性图. 例如 Neo4j[10]、 A1[12]、NebulaGraph[11] 和 IBM Db2 Graph[13] 等知名图数据库. 虽然这些系统能够有效地管理图数据, 但通常缺乏对图拓扑的高效访问. Grasper[17] 是一个利用 RDMA 技术提高性能的图数据库, 与 TAG 类似, 它也采用了拓扑和属性分离的策略. 不同的是, 在 TAG 中并不需要额外的数据结构来存储拓扑, 而是通过索引即拓扑的方式将拓扑存储在系统的索引中, 这种设计降低了同步和存储的开销. 此外, 还有一些研究致力于通过优化索引来存储属性图, 或者在关系数据库中设计专门的表来存储属性图, 增强对图的支持[29].

6 结语

针对当前图存储系统对属性图的支持不足以及无法同时有效支持分析型和事务型负载的问题, 本文设计并实现了一种新颖的图存储系统 TAG. TAG 通过将图的拓扑和属性分别存储在 DRAM 和 PMem 中、将拓扑和索引合二为一、并根据标签组织属性数据等策略, 显著提高了图查询的性能, 特别是增强了属性图上的图遍历性能. 该系统不仅支持属性图上的分析型负载, 还能够支持事务型负载, 具有较好的通用性和扩展性. 实验结果充分证明了我们设计的有效性, TAG 为图的存储提供了一种有价值的解决方案, 有望应用于更广泛的图系统中.

参考文献

1 Sahu S, Mhedhbi A, Salihoglu S, *et al*. The ubiquity of large graphs and surprising challenges of graph processing. Proceedings of the VLDB Endowment, 2017, 11(4): 420–431. [doi: 10.1145/3186728.3164139]

2 DB-Engines. DB-Engines ranking. https://db-engines.com/en/ranking-categories. (2022-11-23).

3 Gartner. Gartner top 10 data and analytics trends for 2021. https://www.gartner.com/smarterwithgartner/gartner-top-10-data-and-analytics-trends-for-2021. (2022-01-03).

4 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. [doi: 10.14778/3384345.3384351]

5 Malewicz G, Austern MH, Bik AJC, *et al*. Pregel: A system for large-scale graph processing. Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. Indianapolis: ACM, 2010. 135–146.

6 Zhu XW, Han WT, Chen WG. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. Proceedings of the 2015 USENIX Annual Technical Conference. Santa Clara: USENIX, 2015. 375–386.

7 Kyrola A, Blelloch GE, Guestrin C. GraphChi: Large-scale graph computation on just a PC. Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation. Hollywood: USENIX, 2012. 31–46.

8 Kumar P, Huang HH. GraphOne: A data store for real-time analytics on evolving graphs. ACM Transactions on Storage, 2020, 15(4): 29.

9 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.

10 Neo4j. https://neo4j.com/. (2023-01-03).
11 VESOFT. NebulaGraph. https://nebula-graph.com.cn. (2023-01-03).
12 Buragohain C, Risvik KM, Brett P, *et al*. A1: A distributed in-memory graph database. Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Portland: ACM, 2020. 329–344.
13 Tian YY, Xu EL, Zhao W, *et al*. IBM Db2 Graph: Supporting synergistic and retrofittable graph queries inside IBM Db2. Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Portland: ACM, 2020. 345–359.
14 Venkataramani V, Amsden Z, Bronson N, *et al*. TAO: How Facebook serves the social graph. Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. Scottsdale: ACM, 2012. 791–792.
15 Robinson I, Webber J, Eifrem E. Graph Databases: New Opportunities for Connected Data. Sebastopol: O 'Reilly, 2015. 10–11.
16 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.
17 Arulraj J, Levandoski J, Minhas UF, *et al*. BzTree: A high-performance latch-free range index for non-volatile memory. Proceedings of the VLDB Endowment, 2018, 11(5): 553–565. [doi: 10.1145/3187009.3164147]
18 Yang J, Kim J, Hoseinzadeh M, *et al*. An empirical guide to the behavior and use of scalable persistent memory. Proceedings of the 18th USENIX Conference on File and Storage Technologies. Santa Clara: USENIX, 2020. 169–182.
19 Xiang LF, Zhao XS, Rao J, *et al*. Characterizing the performance of intel optane persistent memory: A close look at its on-DIMM buffering. Proceedings of the 17th European Conference on Computer Systems. Rennes: ACM, 2022. 488–505.
20 Graefe G. Modern B-tree techniques. Foundations and Trends in Databases, 2011, 3(4): 203–402.
21 Leis V, Haubenschild M, Neumann T. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. IEEE Data Engineering Bulletin, 2019, 42(1): 73–84.
22 Levandoski JJ, Lomet DB, Sengupta S. The Bw-Tree: A B-tree for new hardware platforms. Proceedings of the 2013 IEEE International Conference on Data Engineering. Brisbane: IEEE, 2013. 302–313.
23 Wang ZQ, Pavlo A, Lim H, *et al*. Building a Bw-Tree takes more than just buzz words. Proceedings of the 2018 International Conference on Management of Data. Houston: ACM, 2018. 473–488.
24 Szárnyas G, Waudby J, Steer BA, *et al*. The LDBC Social Network Benchmark: Business Intelligence Workload. Proceedings of the VLDB Endowment, 2022, 16(4): 877–890. [doi: 10.14778/3574245.3574270]
25 Erling O, Averbuch A, Larriba-Pey J, *et al*. The LDBC social network benchmark: Interactive workload. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. Melbourne: ACM, 2015. 619–630.
26 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, 2016. 301–316.
27 Fan WF, Jin RC, Liu MY, *et al*. Application driven graph partitioning. Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Portland: ACM, 2020. 1765–1779.
28 De Leo D, Boncz P. Teseo and the analysis of structural dynamic Graphs. Proceedings of the VLDB Endowment, 2021, 14(6): 1053–1066. [doi: 10.14778/3447689.3447708]
29 Sun W, 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.

(校对责编: 孙君艳)