LMSA：NVM 环境下高性能动态图处理数据结构

祝 贺 华强胜 金 海 廖小飞

（大数据技术与系统国家地方联合工程研究中心 武汉 430074）

（服务计算技术与系统教育部重点实验室 武汉 430074）

（集群与网格计算湖北省重点实验室 武汉 430074）

（华中科技大学计算机科学与技术学院 武汉 430074）

摘 要 在过去十数年，来自网络与社交网络的图信息量在急剧增长，这种本质上动态变化的图对存储、分析与处理的实时性需求越来越高. 新兴的非易失性内存（Non-Volatile Memory，NVM）技术具有高密度、高可扩展性和接近零待机功耗的优点，同时由于字节寻址等特性被认为是替代 DRAM 的潜在候选者，它们可以满足动态图信息快速增长的存储与处理要求. 然而，由于 NVM 的硬件限制和数据一致性要求，传统动态图数据结构在 NVM 环境下效率低下. 为了解决 NVM 环境下动态图数据结构存在的读写不对称和耐久性低等问题，本文设计与实现了层级合并排序数组（Level Merge Sorted Array，LMSA），它是一种支持在对数时间内同时完成读与写操作的动态图数据结构，它使用层级数组存储动态图中边信息来提升查询速度与减少因动态图数据结构性质维护而产生的写次数. 为了低开销地保证数据一致性，LMSA 利用无日志记录一致性方案进行插入、删除和更新等操作. 在配置了英特尔傲腾持久内存 DCPMM（Intel Optane DC Persistent Memory Module）机器上的实验结果表明，与最新的动态图数据结构 Stinger 和 GraphTinker 相比，LMSA 插入操作吞吐量是 Stinger 的 4.3～12.6 倍，是 GraphTinker 的 1.4～4.35 倍，其删除操作吞吐量是 Stinger 的 5.7～20.1 倍，是 GraphTinker 的 1.4～4.58 倍.

关键词 动态图；非易失性内存；数据结构；写优化；崩溃一致性

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2022.01446

LMSA：A High Performance Dynamic Graph Processing Data Structure for Non-Volatile Memory Systems

ZHU He HUA Qiang-Sheng JIN Hai LIAO Xiao-Fei

（*National Engineering Research Center for Big Data Technology and System*，*Wuhan* 430074）

（*Service Computing Technology and System Lab*，*Wuhan* 430074）

（*Cluster and Grid Computing Lab*，*Wuhan* 430074）

（*School of Computer Science and Technology*，*Huazhong University of Science and Technology*，*Wuhan* 430074）

Abstract In the past ten years，the information of graphs from the Web and social networks has been increasing rapidly，and the demand of dynamic graphs for real-time storage，analysis and processing is getting higher and higher. It will bring great challenges such as low throughput and difficulty in maintaining graph properties under random edge updates. Emerging Non-volatile Memory（NVM）storage technologies have the advantages of high density，high scalability and near-zero standby power. As a result，they are considered as potential candidates to replace DRAM due to features such as byte-addressing. They can meet the rapidly increasing storage and processing requirements of dynamic graph information. However，the current state-of-the-art

收稿日期：2020-12-31；在线发布日期：2021-09-26. 本课题得到国家重点研发计划项目（2018YFB1003203）和国家自然科学基金项目（61972447，61832006）资助. **祝 贺**，硕士研究生，主要研究方向为 NVM 环境下高性能数据结构. E-mail：hezhu@hust.edu.cn. **华强胜**（通信作者），博士，研究员，中国计算机学会（CCF）会员，主要研究领域为并行与分布式计算理论与算法. E-mail：qshua@hust.edu.cn. **金 海**，博士，教授，中国计算机学会（CCF）会士，主要研究领域为计算机体系结构、计算系统虚拟化、集群计算和云计算、网络安全、对等计算、网络存储与并行 I/O 等. **廖小飞**，博士，教授，中国计算机学会（CCF）杰出会员，主要研究领域为内存计算、运行时系统、图计算等.

data structures for dynamic graph processing rely on static graph representation models such as adjacency lists of edge blocks when updating graphs. They either have high latency in data query in NVM environments or have serious data movement overheads, which lead to poor update throughput. At the same time, NVM provides non-volatility for main memory data and also needs to consider data crash consistency. It can use logging or Copy on Write (CoW) to ensure data crash consistency greater than the atomic write size of a classic CPU. However, logging and CoW will lead to additional write operations, thereby increasing the overheads of NVM applications. In order to solve the issues of dynamic graph data structures in the NVM environment, Level Merge Sorted Array (LMSA) is proposed in this paper, which can support the optimization of reading and writing performance. It employs a hierarchical array to store dynamic graph edges to improve the speed of query and to reduce the times of writing caused by the maintenance of dynamic graph data structure properties. The LMSA can finish the query operation in $O(\log^2 d_{\max})$ time, complete each insertion operation in $O(\log d_{\max})$ time, and complete each deletion and update operation in $O(1)$ time where $d_{\max}$ means the maximum vertex degree. Based on the idea of LMSA, the proposed adaptive basic array size LMSA can further improve the performance of the LMSA by the tradeoff between space overhead and query latency, which can compress the space to further improve the space utilization and to reduce the NVM read and write times. In order to further improve the space utilization of the LMSA, due to the strict order characteristics of the LMSA level arrays, this paper uses Elias-Fano encoding to implement Compressed-LMSA (C-LMSA). To guarantee consistency with low overheads, LMSA leverages log-free consistency schemes for insertion, deletion, and update operations. Based on the experimental results conducted on the machine equipped with DCPMM (Intel Optane DC Persistent Memory Module), LMSA provides a 4.3x to 12.6x insertions throughput improvement of Stinger and 1.4x to 4.35x of GraphTinker; it also provides a 5.7x to 20.1x deletions throughput improvement of Stinger and 1.4x to 4.58x of GraphTinker where both are the state-of-the-art dynamic graph data structures. Experiments show that C-LMSA can compress the number of LMSA-level arrays to 9%～42% of the original while reducing the update throughput by an average of 20%.

Keywords　dynamic graphs; non-volatile memory; data structure; write optimization; crash consistency

1　引　言

非易失性存储器(NVM)技术具有字节寻址、大容量、低延迟和低功耗等优点[1-2]，它们可以作为处理 DRAM 技术扩展问题的有效解决方案. 非易失性可以使数据把 NVM 作为持久内存进行存储，以进行快速故障恢复. 最近，英特尔发售了 NVM 技术第一个商业产品 Intel Optane DC Persistent Memory Module(以下简称 DCPMM)①，这是第一个针对主流计算机系统的商用 NVM 解决方案. 如表 1 所示，我们通过基准程序[3]比较 DCPMM 与 DRAM 的性能差异，这些差异符合以前研究中关于 NVM 的许多假设. 与 DRAM 相比，DCPMM 具有大约 3 倍的读取延迟和写入延迟，其读写带宽约是 DRAM 的 1/3 和 1/6. DCPMM 的 I/O 性能表现均优于现有的非易失性存储器(例如 Disk 和 SSD 等[4]).

表 1　DRAM 与 DCPMM 性能差别

	随机读延迟/ns	随机写延迟/ns	随机读带宽/(GB/s)	随机写带宽/(GB/s)
DRAM	101.2	104.8	18.3	14.1
DCPMM	303.5	308.0	6.6	2.4

图(Graphs)已经被广泛用于表示不同领域的信息. 无论是生物网络结构[5]，还是社交网络实体间关系[6]，或者是 Web 图中网页之间链接[7]等，都可

① https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html, 2019

以使用图表示方法进行相关数据的存储与处理. Google 知识图从 2008 年的一万亿个页面增长到 2013 年的三十万亿个页面,在短短五年内增长约 30 倍[8]. Facebook 在 2020 年最新的统计信息显示每日活跃用户数量超过 17.3 亿①. 这种动态变化的图每天将产生海量的数据与关系更新,它们将带来随机边更新模式下吞吐量低和图性质维护困难等巨大挑战[9-11]. 人们尝试设计高性能内存计算(In-Memory Computing)数据结构来解决动态图处理中的问题[12-13].

已有图数据结构(如 Stinger[14]、GraphTinker[8]或混合图[15]等)实现了动态图上的高存储效率、高吞吐量和高扩展性. 它们为维护动态图数据结构性质一般使用快照技术(Snapshot)、日志技术(Log)或原数据结构更新等技术. 然而这些技术将带来内存空间开销不断增大、空间利用率和查询性能逐渐降低等问题. 人们一方面使用轻量级快照或日志技术应对这一问题[16],但是为了将图信息持久化,这些快照或日志将保存在 SSD 或 Disk 等慢速的非易失性存储介质中,这可能引发图处理 I/O 瓶颈. 另一方面可在 DRAM 中维护图的单个可变副本,已有几种动态图处理框架[14,17-18]采用这种方式. 然而它们要求实时查询操作和更新操作间隔进行,这将带来图实时查询延迟过高、更新操作无法严格顺序执行等问题,从而降低动态图更新吞吐量与扩展性[19].

研究设计 NVM 技术下的高性能动态图处理数据结构是解决海量图数据下 DRAM 技术存储与计算扩展低效等问题的解决方案之一. 由于 NVM 写读延迟比和耐久性都比 DRAM 差,在满足传统 DRAM 动态图数据结构高更新速度与高扩展性要求的同时,写次数优化与空间利用率优化和低开销一致性保障也是亟待解决的问题.

为了解决这些问题,本文设计了 NVM 环境下权衡读写优化的动态图处理数据结构 LMSA. LMSA 关键思想是将图中每一个顶点的邻接顶点保存在层级排序数组中,数组内元素自下而上进行移动合并,严格要求第 $i+1$ 层数组大小是第 $i(i\geq 1)$ 层数组大小 2 倍,这意味着层级数组高度最大为 $O(\log d_{\max})$($d_{\max}$为图顶点最大度数). LMSA 层级数组都是有序排列,因此可以使用二分搜索(Binary Search)方法在接近对数时间内对图中邻接顶点信息进行查询;除了一致性要求和插入操作将产生常数写次数,在不同层级数组间的元素只有在合并过程中才会产生对 NVM 级联写入操作,合并操作可顺序执行且元素最高可到达 $O(\log d_{\max})$层级数组中,因此在加速更新同时,LMSA 能够达到对数复杂度写次数优化;对数时间开销的读写操作能有效提升动态图处理更新速度. LMSA 以数组形式存储动态图信息,这能够有效降低指针空间开销,从而提高动态图处理时空局部性和提高空间利用率. 因为 LMSA 层级数组严格有序的特性,本文使用 Elias-Fano 编码对有序数组进行压缩处理,它进一步提高 LMSA 的空间利用率. 为了低开销保证数据一致性,LMSA 利用无日志记录一致性方案进行插入、删除和更新等操作.

LMSA 与传统静态图数据结构邻接链表(Adjacency List, AL)、压缩稀疏行(Compressed Sparse Row, CSR)和边列表(Edge List, EL)等对比,它解决了 AL 查询性能低和局部性差等问题;解决了 CSR 在动态扩展过程中数据移动导致的写次数开销高等问题;解决了 EL 查询性能低和写次数开销高等问题. 与最新的动态图数据结构 Stinger 和 GraphTinker 对比,它解决了 Stinger 因依赖 AL 查询效率低和使用边类型数组(Edge Type Array, ETA)索引空间开销大问题;解决了 GraphTinker 哈希计算开销大与哈希碰撞产生额外开销等问题.

为了验证 LMSA 的性能,本文在配置 DCPMM 机器上进行实验评估. 与 Stinger 相比,LMSA 插入操作吞吐量平均提升 4.3~12.6 倍,与 GraphTinker 相比,LMSA 插入操作吞吐量平均提升 1.4~4.35 倍. 对于有 30 亿条边的图,本文使用 Elias-Fano 编码的 C-LMSA 仅仅需要 20.8 GB 内存空间,而 Stinger 每条边需要 140 多字节开销[20],占用 180.3 GB 内存空间. 本文还分析 LMSA 在动态图插入与删除过程带来的读写开销与复杂度,其综合性能远优于已有静态图数据结构. 总的来说,本文主要贡献可以包括如下三点:

(1) 读写优化的动态图数据结构 LMSA. 我们设计的 LMSA 能够做到 $O(\log^2 d_{\max})$时间查找到要查询的边、以 $O(\log d_{\max})$写次数完成每次插入操作和以 $O(1)$写次数完成每次删除与更新操作. 我们在 LMSA 的思想基础上完成自适应基础数组大小的 LMSA 和能够压缩空间的 C-LMSA,进一步提高空间利用率和减少对 NVM 读写操作.

(2) 低开销的数据一致性保障. LMSA 利用无日志记录一致性方案进行插入、删除和更新操作. 通过利用最多两次不大于存储指令大小的原子写操作

① https://zephoria.com/top-15-valuable-facebook-statistics/

来修改层级数组的标志位，以确保数据结构的一致性，而无需昂贵的日志记录等方式.

(3) 真实世界(Real-world)系统实现与评估. 我们使用配置 DCPMM 的真实世界系统实现 LMSA 与 C-LMSA 数据结构，并对它们的查询、插入、删除与更新操作以及空间开销等进行综合实验评估. 通过与 AL、CSR 和 EL 等静态图数据结构和最新的动态图数据结构 Stinger 和 GraphTinker 评估对比，验证了 LMSA 的高效性.

2　相关工作

本节首先介绍非易失性存储器技术与相关数据结构，然后介绍图模型和最新的静态与动态图数据结构，最后分析现有图数据结构对 NVM 写次数产生的影响.

2.1　NVM 器件与数据结构

相变存储器(Phase Change Memory，PCM)、电阻随机存取存储器(Resistive Random Access Memory，ReRAM)、自旋转随机存取存储器(Spin-Transfer Torque RAM，STTRAM)和 Intel 3D Xpoint① 等 NVM 技术具有高密度、高可扩展性和低功耗等优点[21]，这使得 NVM 成为替换 DRAM 的潜在候选者. 同时由于 NVM 非易失的特性，它们可以作为持久存储器来代替 Disk 和 SSD[22]，这使得 NVM 可以同时作为内存与持久存储器来代替 DRAM 和 Disk/SSD，从而避免高速 DRAM 与低速 Disk/SSD 可能存在频繁的缓存(Cache)页面(Page)替换 I/O 问题，以提高计算任务的执行速度.

尽管 NVM 有以上诸多优点，但是使用不同材料与技术的 NVM 都存在共有局限性. 首先，它们都具有读写不对称特性，读带宽比写带宽高 3～8x[22]，同时单位写的能量损耗也比读高. 其次，NVM 通常具有有限的写耐久性，例如，PCM 的写次数为 10^7～10^9[23]. 因此在使用 NVM 设计数据结构时应当注意减少写次数，从而提高数据结构性能与提高 NVM 耐久性.

NVM 为主存储器数据提供非易失性的同时需要考虑数据的崩溃一致性[4]. 对于存储(Store)指令，经典 64 位 CPU 原子写大小为 8 个字节，当数据量的大小大于 8 个字节时，顺序写入数据期间如果发生系统故障，其可能导致数据更新不完整和缓存数据不一致的问题. 当计算机回写高速缓存时，数据持久顺序与存储指令的执行顺序可能不同，因此需要使用内存屏障来实现一致性. 为了保证一致性，现有 CPU 提供有关缓存行刷新(例如 clflush，clflushopt 和 clwb 等)和内存屏障(例如 sfence，lfence 和 mfence 等)的指令. 用户可以使用日志记录或 CoW 来确保大于 8 个字节的数据崩溃一致性[3,23]. 但是，日志记录和 CoW 会生成额外的写操作，从而增加 NVM 应用程序的开销[3,20]. 在 LMSA 的实现中，我们使用 PMDK② 提供的接口在计算机中执行 clwb 和 sfence 指令，以将数据持久存储到 NVM 中.

现有 NVM 数据结构优化主要基于树(Tree)的索引结构[24-25]以高效地适应 NVM 存储系统，从而提高索引性能. 还有一类是基于散列(Hash)索引结构[4]提供快速的查找响应时间. 目前已有关于演化图(Evolving Network)在 NVM 环境下研究工作主要是 NVGRAPH[26]，它主要解决 DRAM 与 NVM 混合内存下的 DRAM 临时数据可能导致的崩溃一致性问题. NVGRAPH 考虑将至少一个数据版本存储在 NVM 中，以便在系统故障时进行恢复，将另一个版本同时存储在 DRAM 和 NVM 中，以减少 NVM 可能诱发较高的读写操作延迟. NVGRAPH 没有考虑单独使用 NVM 作为主存时如何优化动态图更新过程带来的吞吐量与扩展性等问题.

2.2　静态图数据结构

本小节首先给出本文使用的符号表示及含义(如表 2 所示). 我们假设本文所有查询输入的图最多有 n 个顶点与 m 条边，且图最大顶点度数为 $d_{\max}$. 然后本小节总结静态图常用数据结构，包括邻接矩阵、邻接链表、边列表和 CSR 等读开销与空间复杂度. 它们作为基础组件被广泛用于动态图数据结构中[9,14]，它们的结构如图 1 所示.

表 2　符号表示及含义

符号	含义
G	图
$V=\{v_1,v_2,v_3,\cdots,v_n\}$	顶点集合
$E=\{e_1,e_2,e_3,\cdots,e_m\}$	边集合
$e=(u,v,w)$	一条顶点 u 指向 v 权重为 w 的边
N_v	顶点 v 相邻的一组顶点
d_v	顶点 v 的度数
$d_{\max}$	图中最大的顶点度数
d_{avg}	图平均顶点度数
$Id_{\max}$	图邻接顶点最大编号
$\|V\|$	图中顶点个数 n
$\|E\|$	图中边个数 m
Av,Av_i,Av_{ij}	分别表示 LMSA 中层级数组，Av 中第 i 层数组以及 Av_i 中第 j 个元素

① https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html

② https://pmem.io/pmdk/

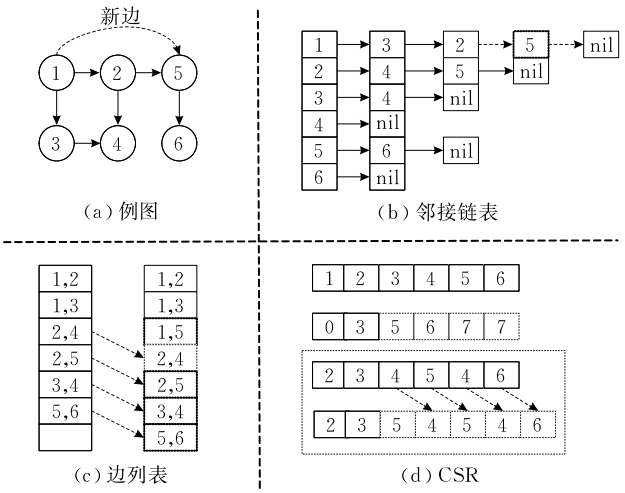


图 1 静态图结构与数据移动(虚线箭头指向的虚线方块表示移动后的数据位置,nil 表示链表末尾)

(1) 邻接矩阵 AM(Adjacent Matrix). 由包含图中边的二维矩阵表示. 此模型提供 $O(1)$ 边查询、插入和删除和更新操作的读时间复杂度. 但是其空间开销达到 $O(n^2)$,对于稀疏或较稀疏图将带来巨大空间浪费,使得空间利用率急剧下降.

(2) 邻接链表 AL(Adjacent List). 每个顶点 u 都有对应的邻接列表 AL_u,它使用指针指向与顶点 u 相邻的顶点 N_u,然后这些顶点将按照链表的方式进行保存,通常指针数量与相邻顶点数量相等. AL 的空间开销为 $O(n+m)$ 和 $O(m)$ 数量级的额外指针开销. 在查询、插入和删除边 e 的过程时会扫描 AL_u,他的读时间复杂度是 $O(d_{\max})$.

(3) 边列表 EL(Edge List). 可以由数组方式保存边 e 的集合,其空间开销最低,严格等于 $O(n+m)$. 边列表分为有序边列表和无序边列表. 有序边列表查询、插入与删除边 e 读时间复杂度都是 $O(\log m)$,无序列表读时间复杂度是 $O(m)$.

(4) 邻接数组(如 CSR 等). 邻接数组通常使用 CSR 进行表示,每个数组通常按顶点 Id 排序. 包含一个对每个相邻数组都有偏移量(或指针)的结构. 由于 CSR 往往由有序数组保存,其读时间复杂度为 $O(\log d_{\max})$.

2.3 动态图数据结构

现有最新动态图数据结构主要包括 Stinger[14]、GraphTinker[8] 等. Stinger 是基于 AL 的内存数据结构且实现并发动态图处理框架. Stinger 每个顶点相邻边是由链表构成的连续块. 每条链表的块除了最后一块其余大小均相同. 顶点可以具有不同类型相邻边. 一个块只包含一种类型边. 使用 AL 意味着 Stinger 对边查询过程需要扫描整个 AL,这将导致查询性能急剧下降. 因此 Stinger 提供 ETA 索引结构以实现快速查找,但是由于 ETA 索引数量有限,且需要遍历扫描,Stinger 依然无法很好解决查询速度慢问题,同时 ETA 会导致空间利用率降低. GraphTinker 主要从更新吞吐量与可扩展性角度优化动态图处理数据结构. 它在 Stinger 的基础上 SGH(Scatter-Gather Hashing)方案与粗粒度(Coarse)AL 压缩和混合模式动态图方案以提高动态图处理性能,但是 SGH 会引入哈希计算开销大与哈希冲突等问题.

还有一类使用树结构保存动态图信息,比如纯函数压缩搜索树 C-Tree(Compressed purely-functional search Tree)[20]. 函数搜索树是一种只能由数学函数表示的树数据结构,它保证数据结构输入对应输出的不可变性(因为数学函数必须保证相同输入将返回相同输出结果,而与数据结构相关的任何状态无关). 这种计算方式主要基于 Hash 取余(mod)思想实现,它可能导致严重的 Hash 碰撞问题. 在 C-Tree 中,一个纯函数树结构存储一组顶点(顶点树),每个顶点边被存储在与自身关联的 C-Tree 中. 图详细信息存储在顶点树中,以便可以在常数时间内查询到图详细属性,例如边和顶点的总数类型等. 类似地,可以扩充树以存储其他扩展属性(例如权重等). 对于在图上运行的算法(例如 BFS 和 DFS 等),C-Tree 会使用数组直接存储指向顶点的指针预处理快照,这样能够保证不查询顶点树信息直接访问到所有顶点,但是它将带来巨大的指针空间开销.

2.4 现有图数据结构对 NVM 写入次数的影响

DRAM 在设计动态图数据结构时主要考虑两个性能参数,即快速更新与高可扩展性[8]. 在 NVM 上设计动态图数据结构时,除了考虑 NVM 的读写不对称特性和有限写入耐久性,还应该考虑 NVM 写次数. 本小节分析现有图数据结构对 NVM 写次数的影响.

图数据结构性质维护过程中不同操作将带来不同写次数. 我们假设对图数据结构查询操作不会破坏图原有结构,它们不会产生写次数. 对于更新操作,图数据结构可以在原地址空间对需要更新的数据进行更改,这将产生 $O(1)$ 的写次数. 因为顶点插入与删除相对于边插入与删除比率十分小[26],因此本文主要考虑边插入与删除操作产生的写次数. 图 1 显示在插入一条新边 $e(1,5)$ 时 AL、EL 与 CSR 的数据移动. 在 AM 中,因为插入与删除操作都是

在矩阵中完成，因此只需要 $O(1)$ 的写入次数. 在 AL 中，假设维护一个指向尾部的链表指针，插入操作只需要将新边插入尾部即可，它需要 $O(1)$ 的写入次数. 对于删除操作，可以将 AL 中要删除顶点的上一个顶点指针指向下一个顶点，故也需要 $O(1)$ 的写入次数. 无序 EL 可以让新加入的边插入到末尾，仅需要 $O(1)$ 的写入次数，但是删除操作因为要维护 EL 的数组结构，需要将删除边的后续边往前移动，此时产生 $O(m)$ 的写次数. 有序 EL 需要将边插入到确定顺序的位置中，这将导致插入边的后续边往后移动一步，将产生 $O(m)$ 写次数. 删除操作与插入操作是一个相反过程，也将产生 $O(m)$ 的写次数. 由于 CSR 以有序数组的方式保存顶点元素，因此在插入过程中需要将顶点插入确定顺序位置中. 如果将 CSR 邻接顶点保存在一个数组中，这将带来 $O(m)$ 的写次数. CSR 的删除操作同理于插入操作也将产生 $O(m)$ 写次数.

图数据结构动态维护的相关复杂度如表 3 所示. Stinger 与 GraphTinker 等动态图数据结构基于多种静态图组合数据结构实现，因此它们的写入次数介于组合数据结构之间，受组合静态图数据结构的复杂度上界影响.

表 3　图数据结构读写开销与空间复杂度对比

数据结构	查询操作		插入操作		删除/更新操作		空间复杂度
	读开销	写开销	读开销	写开销	读开销	写开销	
AM	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
AL	$O(d_{\max})$	$O(1)$	$O(d_{\max})$	$O(1)$	$O(d_{\max})$	$O(1)$	$O(n+m)$
排序 EL	$O(\log m)$	$O(1)$	$O(m)$	$O(m)$	$O(m)$	$O(m)$	$O(m)$
无序 EL	$O(m)$	$O(1)$	$O(m)$	$O(1)$	$O(m)$	$O(m)$	$O(m)$
CSR	$O(\log d_{\max})$	$O(1)$	$O(m)$	$O(m)$	$O(m)$	$O(m)$	$O(n+m)$
LMSA	$O(\log^2 d_{\max})$	$O(1)$	$O(\log^2 d_{\max})$	$O(\log d_{\max})$	$O(\log^2 d_{\max})$	$O(1)$	$O(n+m+\log d_{\max})$
C-LMSA	$O(\log(Id_{\max}/d_v)\cdot \log d_{\max})$	$O(1)$	$O(\log(Id_{\max}/d_v)\cdot \log d_{\max})$	$O(\log d_{\max})$	$O(\log(Id_{\max}/d_v)\cdot \log d_{\max})$	$O(1)$	—

3　LMSA 的设计与实现

本节介绍 Level Merge Sorted Array(LMSA)，它是一种能够同时在对数时间内完成读写操作的高性能动态图处理数据结构. 它使用层级数组存储动态图中边信息来提升查询速度与减少因动态图数据结构性质维护而产生的写次数. 我们首先在 3.1 节介绍 LMSA 基础数据结构，然后在 3.2 节介绍 LMSA 的查询、合并、插入、删除和更新等基础操作，同时介绍利用无日志记录的一致性方案进行插入、删除和更新等操作；在 3.3 节与 3.4 节我们将在 LMSA 的基础上介绍自适应基础数组大小 LMSA 和压缩编码的 C-LMSA，进一步减少对 NVM 读写操作与增大空间利用率；接着我们在 3.5 节介绍使用细粒度锁实现的并发 LMSA；最后在 3.6 节我们对 LMSA 与 C-LMSA 读写复杂度给出分析与证明.

3.1　LMSA 数据结构

在图 G 中每一个顶点 v 都将维护一个 LMSA，它本质是一个倒阶梯型二维矩阵. 如图 2 所示矩阵每一层都是一个给定大小的数组，数组保存顶点 v 的邻接顶点集 N_v. 图 2 中全填充的阴影部分表示层级数组头，包括层级大小(Level)，是否被加锁(Lock)，数组内包含的元素个数(Number)等信息.

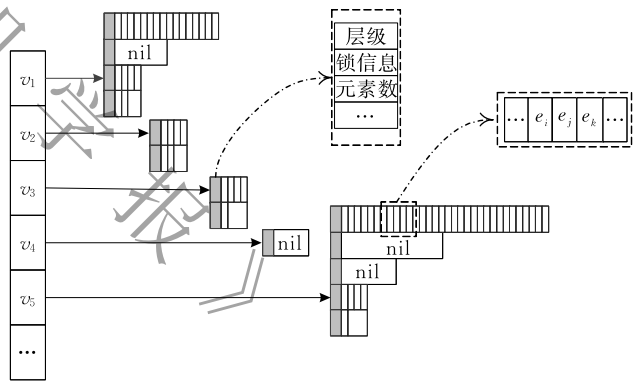


图 2　LMSA 数据结构(左侧由数组保存顶点，右侧由 LMSA 保存顶点的邻接边集合，深色部分保存层级数组的头部信息，包含层级、数组元素数和锁信息，层级数组内保存有序边编号及其它信息，nil 表示指针指向为空)

条纹部分表示层级数组中存在元素，空白部分表示数组中没有元素，仅有一个指向它的指针.

LMSA 是一个增量数据结构，它能够高效完成动态图顶点与边的查询、更新、插入和删除等操作. 它能以 $O(\log^2 d_{\max})$ 的时间查找到要查询的边 e、以 $O(\log d_{\max})$ 的写次数完成每次插入操作和以 $O(1)$ 的写次数完成每次删除与更新操作. 初始化后的 LMSA 只有第 0 层数组，记作 Av_0，被称为基础数组(Basic Array)，数组大小可动态设定，记作 m_size. Av_0 是一个未经过排序的数组，它负责存储动态图

变化时顶点 v 动态插入和更新的邻接顶点.当插入的邻接顶点数量超过 m_size 时,LMSA 会触发数组移动行为.此时首先把 Av_0 中的顶点按照编号进行排序(Sort),然后移动到更高层级数组 $Av_i(i\geq 1)$ 中.LMSA 规定 Av_i 是有序数组,而且 Av_i 第 i 层数组大小始终是第 $i-1$ 层数组大小的 2 倍,这意味着层级数组高度最大为 $O(\log d_{\max})$.一般情况下,Av_i 第 i 层数组元素来自于 $0\sim i-1$ 层数组,即 LMSA 中元素由下往上进行移动,这意味着越高层级的元素存在时间越久.当 Av_{i-1} 元素将要移动到 Av_i 时,如果 Av_i 为空,则直接移动即可.如果 Av_i 的元素已满,此时将触发合并(Merge)行为,它会将 Av_i 与 Av_{i-1} 合并为一个 Av_{i+1} 相等大小的临时排序数组 Av'_{i+1},然后尝试将临时数组元素移动到第 $i+1$ 层数组中.如果 Av_i 已经位于 LMSA 最高层级,LMSA 将申请一个 Av_i 两倍大小新的最高层级数组 Av_{i+1},然后把 Av'_{i+1} 移动到 Av_{i+1} 中.如果 Av_i 没有位于 LMSA 最高层级,则 Av'_{i+1} 元素将要移动到 Av_{i+1} 中,与 Av_{i-1} 元素移动到 Av_i 过程同理.

因为 LMSA 每层数组大小始终为低一层数组大小的两倍,因此它最大层级为 $O(\log d_{\max})$,最多消耗 $O(\log d_{\max})$ 的指针空间指向每一层数组.N_v 中每个邻接顶点在最坏情况下将在每次合并操作过程中产生一次数据移动,只有数据移动过程会产生级联写入.因此 LMSA 插入操作需要的写次数开销仅仅为 $O(\log d_{\max})$.

对于 LMSA 删除操作,本文使用延迟删除技术,每次需要被删除的邻接顶点首先会被标记为已删除状态.然后在合并过程中会判断顶点状态,如果状态显示已经被删除,则不会被合并加入到新的层级数组.因此 LMSA 在删除过程中并不会带来数据的移动,删除操作需要的写次数为 $O(1)$.如果在动态图变化过程中存在大量的删除操作,延迟删除由于邻接顶点仍然保存在 LMSA 中并没有被实际删除,它可能会带来空间利用率逐渐降低的问题,我们使用降级合并机制来应对这一问题.降级合并机制主要思想是当发现最高层与次高层的排序数组中存活顶点数目之和小于等于次高层数组大小时,将会把最高层数组与次高层数组进行合并,然后降级移动到次高层数组中,这样可以始终保证最高层级数组是接近充满.因为最高层数组大小是所有更低层数组大小的总和,因此 LMSA 能够始终保证 50% 以上的空间利用率.

3.2 LMSA 实际操作

由于人们主要关注图中顶点的连接信息[20],因此本文使用边顶点编号与详细信息(权重、时间戳和边属性等)分离的方式存储边信息.这样在 LMSA 操作过程中专注对顶点编号操作,它可以进一步提高缓存中有效数据容量,从而降低缓存不命中(Miss)次数.LMSA 只在顶点 v 每一层数组中保存邻接顶点的编号与指向详细信息的指针.本小节我们将介绍 LMSA 实际操作,包括边查询操作、层级数组合并操作、无日志的边插入操作、无日志的边删除和更新操作等.

为了进一步减少写次数,我们使用一个位数组(称为 BasicBitArray)映射基础数组中每个元素编号,"1"表示有效,"0"表示无效.同时使用位数组(称为 LevelBitArray)映射层级数组的层级标志(包括基础数组)是否有效,"1"表示有效,"0"表示无效.由于层级数组元素数目随着层级增高指数增长,位数组只需要一个原子写大小(8 个字节)就能容纳超过 2^{63} 个元素,其远大于真实世界图中边是数目,因此我们假设 LevelBitArray 大小为一个原子写大小.

3.2.1 查询操作

对于查询操作,LMSA 首先会检查输入边 e 的编号是否合法,然后查询边 e 是否存在,由于 LMSA 基础数组无序存储,因此需要遍历查询边 e 是否存在,如果存在则可直接返回边存在的结果.最后 LMSA 会在每一层对有序数组使用二分搜索方法查找待查询边 e,当找到目标或每一层查询都未找到目标时算法结束.如算法 1 所示,如果在每一层数组中都没找到边 e,这表明该边不在 LMSA 中,将返回布尔(Boolean)类型值 FALSE.

算法 1. $FindEdge(Av,e,m\_size)$ //在层级数组中查找边 e

输入:层级数组(Av)、边(e)、基础数组大小(m_size)
输出:布尔值 TRUE 或 FALSE
1. 检查边编号是否合法
2. FOR $j=0$; $j<m\_size$; j++ DO
3. IF Av_{0j} 等于 e THEN
4. RETURN TRUE
5. END IF
6. END FOR
7. FOR $i=1$; $i<Av$ 数组个数; i++ DO
8. IF 二分搜索层次数组 Av_i 找到边 e THEN
9. RETURN TRUE
10. END IF
11. END FOR
12. RETURN FALSE

3.2.2　合并操作

LMSA 合并操作是维护层级数组变化与动态图高效更新的关键操作，它使用一种递归方式完成. 在 LMSA 合并操作中，首先会检查第 l 层数组大小与待合并数组大小是否合法. 如果第 l 层数组为空(即元素个数为 0)，可直接将待合并数组元素移动到其中. 如果第 l 层数组已满，将需要与待合并数组按照递增顺序进行合并，然后生成一个新数组，接着将递归对新数组与 LMSA 第 $l+1$ 层数组执行合并操作. 如算法 2 所示，如果正确执行完合并操作将返回布尔值 TRUE.

算法 2.　*Merge*(Av,B,*size*,l) //将数组 B 合并到 LMSA 层级数组中

输入：层级数组(Av)、待合并数组(B)、待合并数组大小(*size*)、层级数组层数(l)

输出：布尔值 TRUE 或 FALSE

1. 检查 *size*、l 是否合法
2. IF Av_l 为空 THEN
3. 　将数组 B 中的元素移动到数组 Av_l 中，并清空数组 B
4. 　clwb(LevelBitArray[i]); sfence() //修改第 i 层标志位，且使用 sfence 设置内存屏障
5. 　RETURN TRUE
6. ELSE
7. 　合并数组 B 和数组 Av_l 成为一个新的数组 C
8. 　递归调用算法 2 *Merge*(Av,C,*size*×2,$l+1$)
9. 　RETURN TRUE
10. END IF

3.2.3　插入操作

当 LMSA 对一条边 e 进行插入操作时，首先会检查边 e 的合法性，然后检查其是否存在，如果边 e 不存在，接下来将尝试直接将边 e 插入到 LMSA 基础数组尾部. 如果基础数组已满，LMSA 将触发 3.2.2 节算法 2 所示的数组合并操作. 为了低开销地保证 NVM 中数据一致性，以下我们将介绍无日志插入方案.

无日志插入操作. 首先检测基础数组是否已满，如果没有满，只需要将元素插入到基础数组中，此时将产生两种动作.

(1)无合并动作. 此时插入不会产生任何移动就能将新元素插入到基础数组中. 在此情况下，我们首先将新元素写入基础数组，然后将 BasicBitArray 对应位置标志从“0”改为“1”. 通过 sfence 确保写入元素和更改标志的顺序. 尽管新元素一般大于 8 个字节，但是写入元素时无需进行日志记录或 CoW 操作，因为该元素在标志设置为“1”之前一直无效. 如果在元素写入期间发生系统故障，该元素可能被部分写入，但是在 BasicBitArray 中无效，因为当前标志位为“0”，基础数组在该位置仍然可用. 因此，当发生系统故障时，基础数组处于一致性状态，即 LMSA 也处于一致状态. 故可以在无日志情况下完成无合并动作的插入操作.

(2)发生合并动作. 当发生合并动作时可能涉及多个层级数组之间的数据移动. 我们与没有合并动作设计相似，首先将新元素数组执行 3.2.2 节算法 2 合并操作，最后完全合并成功会将 LevelBitArray 对应层级标志位从“0”更改为“1”(移入数组)或从“1”改为“0”(移出数组). 由于我们始终保存更低层级数组的元素，即使在合并期间发生系统故障，层级数组因为标志位没有改变，数据依然有效. 这种策略会带来一定的空间浪费，因为最高层级数组大小一直有效而且约等于更低层数组元素之和，故在最坏浪费 50% 空间的情况下能够完成合并动作的无日志插入.

算法 3.　*InsertEdge*(Av,e,*m_size*) //在层级数组中插入边 e

输入：层级数组(Av)、边(e)、基础数组大小(*m_size*)

输出：布尔值 TRUE 或 FALSE

1. 检查边的编号是否合法
2. IF 调用算法 1 *FindEdge*(Av,e,*m_size*)结果为 TRUE THEN //边已经存在
3. 　RETURN FALSE
4. END IF
5. IF Av_0 的数组元素个数等于*m_size* THEN //Av_0 中元素已满，触发合并操作
6. 　对 Av_0 进行排序
7. 　调用算法 2 *Merge*(Av,Av_0,*m_size*,1)
8. 　清空数组 Av_0
9. 　clwb(BasicBitArray); sfence() //将 BasicBitArray 标志全部改为“0”
10. 　clwb(LevelBitArray[0]); sfence()
11. END IF
12. 将边 e 添加到 Av_0 尾部空闲位置中
13. clwb(BasicBitArray[x]); sfence() //将 BasicBitArray 插入的第 x 位的标志改为“0”
14. RETURN TRUE

3.2.4　删除与更新操作

删除与更新操作都依赖于算法 1 的查询操作完成. 当查询待删除边 e 存在时，LMSA 会将指向详细边信息的状态改为被删除状态，此时完成延迟删除操作. 之后的动态图更新操作将检查最高一层数组

与次高一层数组未被删除元素之和是否小于等于次高层级数组大小，如果满足则把两个数组元素合并到次高层数组中. 更新操作首先会查询待更新边 e 是否存在，如果存在则更新边详细信息.

无日志删除操作. 删除操作仅需要执行一次原子写操作就可以更改状态位. 删除 LMSA 中元素时，我们会把边或顶点的状态从“1”改为“0”，这意味着此处元素已经处于无效状态，此时可被插入新的元素.

无日志更新操作. 更新 LMSA 中的元素时，如果层级数组中元素在被更新时发生系统故障，将导致数据不一致. 我们首先将待更新的元素写入到基础数组中，操作成功后会将 BasicBitArray 对应标志从“0”更改为“1”，然后再使用一次原子写操作将待更新元素的删除状态改为“0”. 这种操作需要执行两次原子写操作，如果在完成 BasicBitArray 标志修改后发生系统故障，待更新元素的删除状态可能没有改变，但是因为 LMSA 层级数组由下而上移动，这意味着更低层次的数组元素是更新的，当查询出现重复元素时我们得到基础数组的元素有效（即新插入的元素有效）. 故可以使用无日志方式完成更新操作.

3.3 自适应基础数组大小 LMSA

动态图顶点度数存在分布不均匀情况. 当 LMSA 中 Av_0 的 m_size 过小，会使得 Av_0 快速被写满，接着触发频繁的移动与合并操作，从而降低系统性能. 当 Av_0 的 m_size 过大时，对低度数顶点将造成巨大空间资源浪费. 因此为了更进一步提高低度顶点空间利用率和减少高度顶点由于频繁合并产生的写次数，本文将 LMSA 第 0 层基础数组大小设置为自适应变化模式.

如图 3 所示，当新边插入过程中发现顶点对应的 LMSA 层数超过一个给定阈值并且 Av_0 写满时，首先完成算法 3 插入操作，然后尝试将 Av_0 元素排序后与 Av_1 元素一起合并移动到 Av_2 中，此时将 Av_1 作为新的基础数组. 当完成基础数组扩容后，将 Av_0 与基础数组空间释放. 这种方式可以在不额外增加写开销的情况下将 LMSA 转化为自适应基础数组大小 LMSA. 然而 m_size 增大意味着在查询操作过程中对无序数组遍历时间变长，因此 m_size 不能够无限增大，自适应基础数组大小 LMSA 是对基础数组大小与层级数组高度的一种权衡，即对读写开销的一种权衡.

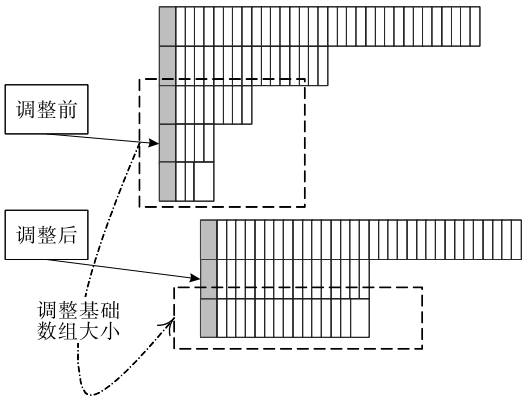


图 3 自适应基础数组大小 LMSA（图示当 LMSA 层级升高时，基础数组大小调整为原来的四倍）

3.4 使用压缩模式的 C-LMSA

3.4.1 Elias-Fano 编码模式

Elias-Fano 编码由 Peter Elias 和 Robert Mario Fano 分别在 20 世纪 70 年代独立提出来，它可以对连续非负整数的单调非递减序列进行压缩编码. 如图 4 所示 Elias-Fano 编码的核心思想是将非负整数二进制表示为高位（High Bits）与低位（Low Bits）两个部分，假设高位位数为 hb，低位位数为 lb. 高位会落在一个大小为 2^{hb} 的桶中，其由一元编码进行编码后储存. 低位直接把所有元素经低位编码合并，最后结果由高位编码加上低位编码所得. Elias-Fano 编码支持常数时间内访问第 i 个元素[27]，同时能够以近常数时间（受序列中最大数值和序列长度的影响，若为层级数组最高一层则需要 $O(\log(Id_{\max}/d_v))$ 的时间）查询到与待查询整数最接近的那个整数. 结合 LMSA 层级数组严格递增特性，我们提出了压缩 LMSA（Compressed-LMSA，C-LMSA）.

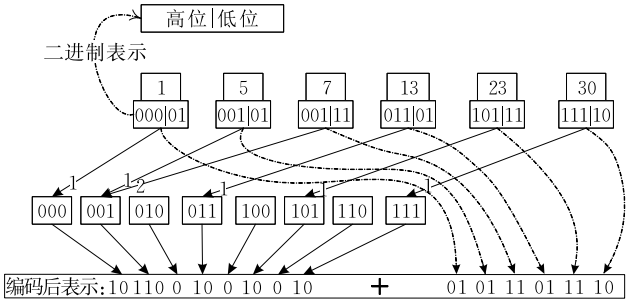


图 4 Elias-Fano 编码模式（图示将有序非负整数序列 1、5、7、13、23 和 30 压缩为最底层的两串数字）

3.4.2 C-LMSA

我们利用 LMSA 中大于第 0 层数组严格递增特性，对每层排序数组进行 Elias-Fano 编码，从而更进一步减少 LMSA 空间开销，这种经过 Elias-Fano 编码压缩后的 LMSA 称为 C-LMSA. 对于查询操

作，C-LMSA 能够充分利用 Elias-Fano 良好查询性能，从而降低层级数组查询时间复杂度. 对于插入操作，基础数组的插入操作与 LMSA 相同. 但是在向上移动过程将触发 Elias-Fano 编码的额外编码开销，同时为了解决合并操作过程解码带来的额外开销，本文并不将 C-LMSA 完全解码后进行合并，而是先在两个经过压缩编码的层级数组提取出最大元素，然后按照排序元素归并的方法同时进行解码与合并操作，最后再共同编码. 这样即使动态图变化过程中触发大量合并操作也不会明显地增加空间开销.

受 Elias-Fano 随机查询时间受整数最大数值影响与现实世界图数据集保存方式[20]的启发，对于无向图，C-LMSA 会在顶点 v 的邻接顶点保存使用比它大的顶点编号，然后将这些邻接顶点编号与顶点 v 做差值进行保存，这样就可以减小 LMSA 排序数组中的最大元素，从而进一步增大压缩率，减少查询时间. 对于有向图，C-LMSA 会在层级数组头部记录该数组第一个元素（最小元素），然后同理进行差值保存. 经过实验评估，对于数据集 100K_100M（见表 4），C-LMSA 压缩率可达到 10%以下.

3.5　并发 LMSA

并发数据结构设计对 NVM 系统扩展到更多数量内核与线程至关重要. LMSA 使用数量有限的关键指针（层级指针）而且每个顶点单独存储，这些特性使得它们不易产生数据冲突，从而使 LMSA 可以通过细粒度锁有效地支持并发读写操作.

在并发 LMSA 中，当不同线程同时读/写同一顶点级别数组时，可能会发生数据冲突. 因此，我们为每个顶点的级别数组分配细粒度锁. 当读或写一个顶点的级别数组时，执行线程首先将其锁定，当完成读/写操作后会释放锁. 为了更进一步提高并发度，我们将空闲线程对级别数组的不同级别执行并发查询操作，由于不同级别的数组在查询期间不会产生数据冲突，我们可以利用无锁操作完成该并发查询操作.

3.6　复杂度分析

定理 1.　LMSA 最多使用 $O(\log^2 d_{\max})$ 的读开销能够完成查询、插入、删除和更新等操作.

证明.　已知 $d_{\max}$ 是动态图中顶点最大的度数，而且 LMSA 最高一级数组包含的元素是其下面所有数组包含元素之和，因此层级数组元素最多为 $d_{\max}/2$. 又因为 LMSA 层级数组是由指数（2 倍）递增的方式组合，因此 LMSA 级数为 $\log d_{\max}$. 当对 LMSA 进行查询、插入、删除和更新操作时会检查元素是否存在，由于层级数组有序特性，可以对每一级数组进行二分搜索方式完成，其最大开销为 $\log d_{\max}$. 故 LMSA 最多使用 $O(\log^2 d_{\max})$ 的读开销能够完成查询、插入、删除和更新等操作.　证毕.

定理 2.　C-LMSA 最多使用 $O(\log d_{\max}\cdot\log(Id_{\max}/d_v))$ 的读开销能够完成查询、插入、删除和更新等操作.

证明.　由定理 1 可知 C-LMSA 级数为 $\log d_{\max}$. 因为 C-LMSA 每一层数组都经过 Elias-Fano 编码进行压缩处理. 文献[27]对 Elias-Fano 编码压缩序列的读开销为 $\log(Id_{\max}/d_v)$，同时 C-LMSA 对于查询、插入、删除和更新等操作会在每一级数组查询压缩序列，故读开销为 $O(\log d_{\max}\cdot\log(Id_{\max}/d_v))$.　证毕.

定理 3.　LMSA 与 C-LMSA 最多使用 $O(1)$ 的写开销能够完成查询、删除和更新等操作. 最多使用 $O(\log d_{\max})$ 的均摊写开销能够完成插入操作.

证明.　LMSA 与 C-LMSA 在查询时不会修改原数据结构，因此写开销为 $O(1)$. 由于删除和更新操作会在原址空间进行，故写开销也为 $O(1)$. LMSA 与 C-LMSA 在插入操作的过程中可能引起基础数组排序与归并操作，写高效（Write Efficient）排序算法[28]可将排序写开销降为 $O(m\_size)$，故均摊在每一个元素中只需要 $O(1)$ 的写开销. 同时有序数组的归并写开销为 $O(d_{\max})$，因此均摊到每一个元素中为 $O(1)$. 因此元素在层级数组的每一级移动开销为 $O(1)$. 由定理 1 和 2 可知 LMSA 与 C-LMSA 的层级为 $O(\log d_{\max})$，故插入写开销为 $O(\log d_{\max})$.　证毕.

4　实验结果与分析

本节将通过实验评估 LMSA 与 C-LMSA 等数据结构的性能. 除非特别说明，评估都是在 NVM 环境下完成.

4.1　实验设置

本文使用的实验服务器配备 27.5 MB 的 L3 缓存、128 GB 的 DRAM 内存和 4 个 256 GB 的 DCPMM 存储器，共计 1TB 的 NVM 存储. DCPMM 有三种可选择执行模式：

(1) 内存模式（Memory Mode）. 内存模式下 DCPMM 会被作为 DRAM 扩展内存使用，DRAM 此时会作为它的更高一级缓存，同时 DCPMM 持久化功能被禁止使用. 可以简单地理解该模式下 DCPMM 被当作 DRAM 内存使用.

(2)直接应用模式(App Direct Mode).该模式提供直接访问(Direct Access)与块存储(Block Storage)两种方式以作为持久存储器供操作系统与应用程序使用.直接访问方式允许 DCPMM 使用字节寻址的方式对应用程序进行操作.块存储方式类似 SSD 使用.

(3)混合模式(Hybrid Mode).混合模式将内存模式和直接应用模式混合使用,它会将一部分 DCPMM 容量用于内存模式使用,剩余容量部分用于直接应用模式使用.

在本文中,NVM 由 DCPMM 在 App Direct Mode 下配置,并与 ext4-DAX 文件系统一起安装.服务器有 2 个 20 核的 Intel Xeon Gold 6230 CPU@2.10GHz 处理器(即 NUMA 节点).由于跨多个 NUMA 节点实验会引入非均匀内存延迟干扰等问题[3],我们在一个 CPU 插槽上执行所有实验.关于 NVM 编程,我们使用 PMDK 将文件映射到程序的虚拟地址空间,并使用加载和存储指令访问 DCPMM.我们使用 clwb 和 sfence 将数据持久保存到 DCPMM 内存中.

数据集.本文使用的真实世界图可由斯坦福大学标准大规模网络数据集[29]获得.由于真实图数据集多是稀疏图,因此本文又通过合成图方式来生成较稀疏图与较稠密图来对实验进行充分评估,合成图可由 Graph500 RMAT 生成器[30]生成,我们生成的 RMAT 合成图是拥有重复边(数目低于 1/20)和随机排列的无向图,由边数除以顶点数可得到图的平均度数.这些数据集属性如表 4 所示.

表 4 待评估图数据集详细信息

图数据集	类型	顶点数	边数
100K_100M	RMAT 合成图	100000	100000000
40M_1B	RMAT 合成图	40000000	1000000000
100M_3B	RMAT 合成图	100000000	3000000000
com-orkut	真实世界图	3072626	117185087
twitter7	真实世界图	41652230	1468365182

4.2 LMSA 与 AL、CSR 及 EL 更新性能对比

本小节将使用数据集 100K_100M 评估 LMSA 与静态图数据结构 AL、CSR 和 EL 等更新性能.我们首先将 100K_100M 前 1 千万条边插入到这四种数据结构中,然后以一百万条边作为一个批次不断按照批次插入到四种数据结构中来模拟动态图边更新过程,最后统计它们在每个批次中的用时(单位:s),将一百万除以用时可以得到更新吞吐量.图 5 展示 LMSA 性能远远优于 AL、CSR 和 EL.与 2.2 节和 2.4 节分析结果一致,在每一批次更新时,EL 由于查询时间的增大会随着存储边的增加线性增加更新用时,CSR 与 EL 由于大量的数据移动开销,更新用时急剧增长.LMSA 在批次增加的过程中用时是缓慢增长的(由于数值过小在图中不易显示,LMSA 更新时间开销按横坐标从左到右的数值分别为 0.44、0.49、0.54、0.6、0.64、0.51、0.74、0.64 和 0.67 s),这意味着其具有良好的稳定性.

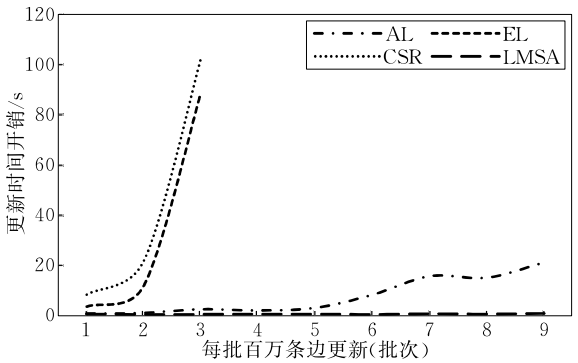


图 5 批次大小为一百万条边时 LMSA 与 AL、CSR 及 EL 的更新时间开销(s)对比

4.3 LMSA、Stinger 和 GraphTinker 性能对比

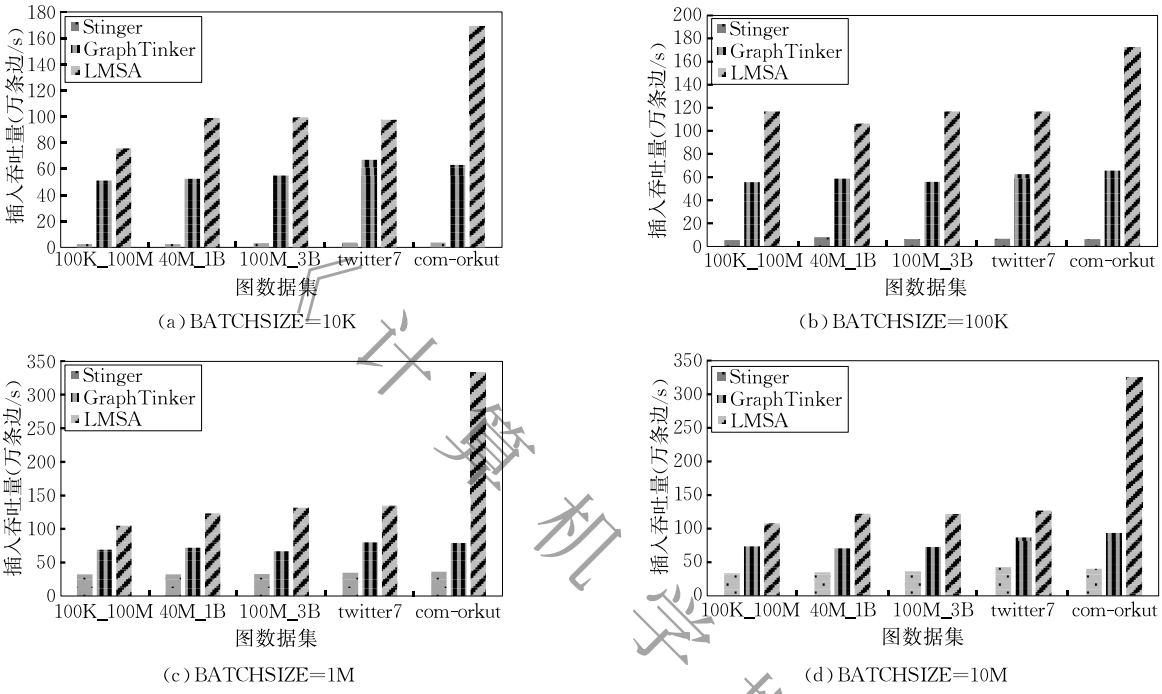
本小节将首先展示使用数据集 100K_100M、40M_1B、100M_3B、com-orkut 和 twitter7 在 NVM 环境下与 Stinger 和 GraphTinker 在不同批次大小下的插入操作性能,然后展示 LMSA 与 Stinger 的删除性能对比.结果显示即使非常大的图和非常大尺寸的批次,LMSA 也能实现很高的吞吐量.最后我们评估 LMSA 在 DRAM 与 NVM 环境下插入吞吐量.我们通过先设定批次大小,然后将边批量插入到三种数据结构来模拟动态图更新过程.

Stinger 是一个开源共享内存动态图数据结构,它同时实现并行动态图处理框架.本文使用 GitHub 上最新 15.10 版本①.Stinger 配置每个 edgeblock 大小为 16.GraphTinker 是在改进 Stinger 基础上引入 SGH 方案与粗粒度 AL(CAL)压缩和混合动态图处理模式实现,我们使用 GraphTinker 最新开源版本②.为了适应 NVM 环境,以上两种数据结构主要通过 PMDK 修改它们内存分配方式,并使用加载和存储指令访问 DCPMM 来进行对比评估.

图 6 展示在不同批次大小(10K、100K、1M 和

① https://github.com/stingergraph/stinger
② https://github.com/Wole308/graphtinker

10M)下的插入吞吐量,将批处理大小除以吞吐量可得执行时间.评估表明 GraphTinker 与预期一样优于 Stinger,原论文[8] GraphTinker 在 DCPMM 中性能下降速度快于 Stinger 下降速度.我们发现在每秒处理百万条边(1M)时三种数据结构插入吞吐量最大.此时 LMSA 插入吞吐量是 Stinger 的 4.3～12.6 倍,是 GraphTinker 的 1.4～4.35 倍.我们发现对于 com-orkut 数据集评估,LMSA 有极大的性能提升.主要原因是该数据集模拟动态插入过程为边顶点序号有序插入.这种有序插入方式将使得 LMSA 在插入操作过程中可以利用良好的空间与时间局部性,因此极大地提升了吞吐量.100K_100M、40M_1B、100M_3B 和 twitter7 这四个数据集模拟了边顶点序号随机情况下的插入吞吐量,在随机插入过程中 LMSA 性能依然优于 Stinger 和 GraphTinker.



(a) BATCHSIZE=10K　(b) BATCHSIZE=100K

(c) BATCHSIZE=1M　(d) BATCHSIZE=10M

图 6　不同 BATCHSIZE 下 LMSA 与 Stinger 和 GraphTinker 在不同数据集上的插入吞吐量(万条边/s)对比

图 7 显示 LMSA 与 Stinger 和 GraphTinker 在五种数据集的删除性能对比.我们设置批次为每秒删除百万条边,我们发现 100K_100M 由于度数最大导致频繁降级合并而吞吐量最低.我们发现在每秒更新百万条边时 LMSA 的插入吞吐量是删除吞吐量的 11%～19%,同时 LMSA 删除吞吐量是 Stinger 的 5.7～20.1 倍,是 GraphTinker 的 1.4～4.58 倍.

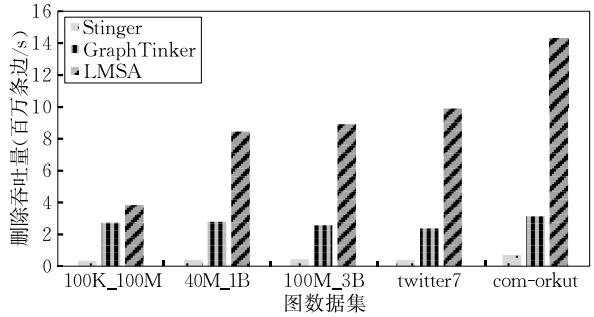


图 7　批处理大小是一百万条边时 LMSA 与 Stinger 和 GraphTinker 的删除吞吐量(百万条边/s)对比

图 8 展示 DRAM 和 NVM 环境下 LMSA 在批次大小为一百万条边时插入吞吐量对比.LMSA_NP 表示关闭持久化功能,LMSA_P 表示开启持久化功能(本节其它部分 LMSA 皆表示开启持久化功能).我们发现 DRAM(关闭持久化功能)插入吞吐量大约是 NVM(开启持久化功能)1.7 倍,这主要是因为访问 DRAM 速度更快.

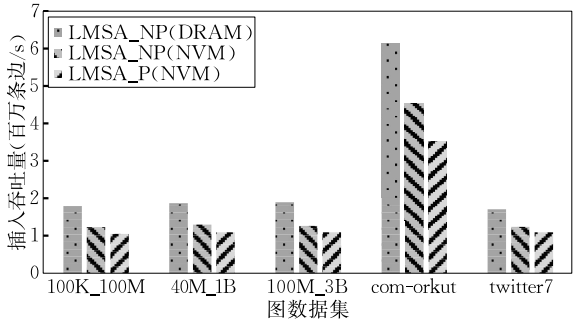


图 8　批次大小为一百万条边时 LMSA 在 DRAM 与 NVM 环境下插入吞吐量(百万条边/s)对比

我们发现 LMSA(关闭持久化功能)插入吞吐量大约是 LMSA(开启持久化功能)1.2 倍,这主要是因为完成持久化功能的 clwb 与 sfence 等原子指令消耗很大计算资源,从而较为严重影响 LMSA 性能.

表 5 显示不同数据集下 LMSA、Stinger 和 GraphTinker 运行时内存开销.可以发现 LMSA 内存开销是 Stinger 的 11%～13%,是 GraphTinker 的 11%～21%.随着边度数的增大,Stinger 内存消耗在增大,故其扩展性较差.GraphTinker 由于使用大量的树结构指针与 Hash 指针,依然存在内存开销大的问题.

表 5 不同图数据集运行时内存占用 (单位:GB)

数据集	Stinger	GraphTinker	LMSA
100K_100M	15.2	13.7	1.4
40M_1B	80.3	66.6	9.8
100M_3B	180.3	156.6	20.8
com-orkut	15.4	13.6	1.5
twitter7	89.8	78.6	12.2

4.4 自适应 LMSA 在不同基础数组大小的测试对比

本小节分别设置 LMSA 在不同基础数组大小下的更新吞吐量.如图 9 所示,LMSA 吞吐量会随着不同基础数组大小的增大先增大然后保持平稳最后可能逐渐下降,这与 3.3 节自适应基础数组大小 LMSA 的分析一致.对比五种数据集,100K_100M 具有最大度数,这将导致 LMSA 可能产生多次合并操作,与此同时,查询时间开销将不断增大,因此其吞吐量最低.

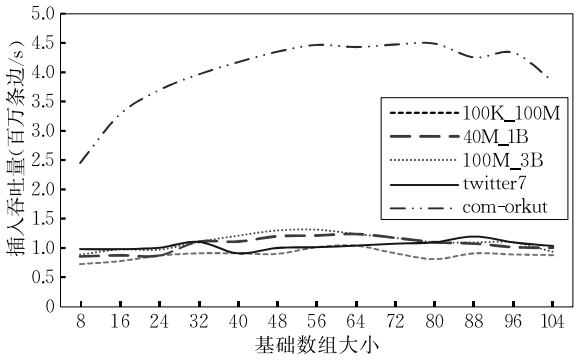


图 9 LMSA 在不同基础数组元素个数下的插入吞吐量

对于数据集 100K_100M、100M_3B 和 com-orkut,我们发现在基础数组大小为 64 时,LMSA 具有最佳吞吐量,主要原因是这些数据集的平均度数小于 64 且分布均匀,因此具有最佳吞吐量.本节验证自适应数组大小 LMSA 性能介于基础数组大小 8 到 64 之间.这种权衡的思想有助于进一步提升 LMSA 的更新性能与提高空间利用率.

4.5 C-LMSA 空间压缩率与时间开销对比

我们使用 Elias-Fano 编码变形方式实现适应于 LMSA 且带有压缩功能的 C-LMSA,在该节中将评估 C-LMSA 的空间压缩率,C-LMSA 与 LMSA 的更新吞吐量.

图 10 展示了 C-LMSA 对层级数组的压缩率在 9%～42%之间.因为 Elias-Fano 编码只对有序整数序列进行压缩编码,因此 C-LMSA 会随着基础数组元素个数的增多压缩效果不断下降.100K_100M 具有最高压缩率,主要原因是该数据集下的 C-LMSA 每条边对比其它数据集具有较大度数.本节验证随着数据集度数的增大,LMSA 中层级数组相较基础数组的比例增大,压缩率将逐渐增大,压缩效果更好.

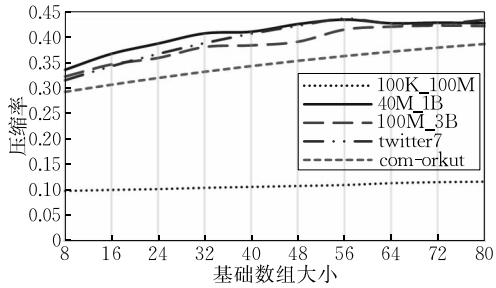


图 10 不同基础数组元素个数下 C-LMSA 的压缩率

图 11 与图 12 分别使用 40M_1B 与 twitter7 数据集展示 LMSA 与 C-LMSA 的插入吞吐量对比.C-LMSA 的插入吞吐量是 LMSA 的 0.66～0.85 倍.

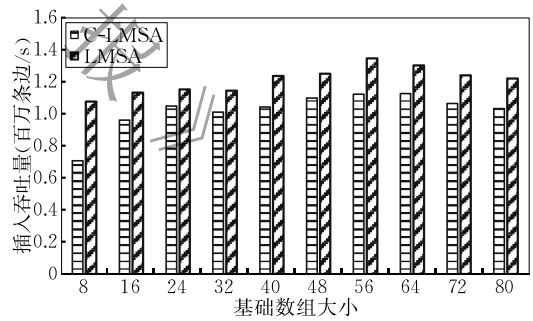


图 11 在 40M_1B 数据集中不同基础数组元素个数下 LMSA 的插入吞吐量

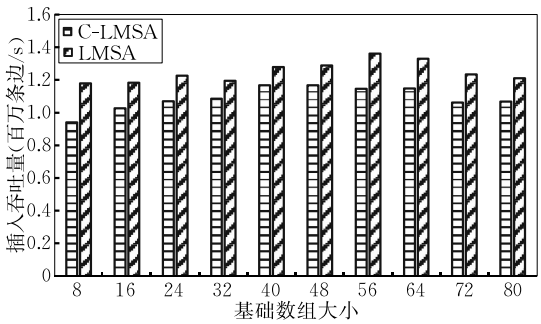


图 12 在 twitter7 数据集中不同基础数组元素个数下 LMSA 的插入吞吐量

我们发现 C-LMSA 性能相较 LMSA 会下降，其主要原因是在构建 Elias-Fano 对象时与访存过程中将需要额外的时间与空间开销. 但是这种性能下降在内存不足的情况可以接受，因此在实际使用 C-LMSA 过程中应当对压缩率提升与性能开销下降做出权衡，从而最大化提升整个动态图处理性能.

4.6　并发 LMSA 插入吞吐量

图 13 展示不同线程数目(2、4、8、12 和 16)下对数据集 40M_1B、100M_3B 和 twitter7 按照每批次处理 100 万条边并发 LMSA 的平均插入吞吐量. 我们发现并发 LMSA 随着线程数量的增加，插入吞吐量保持线性增长，说明它具有良好的并发特性与扩展性.

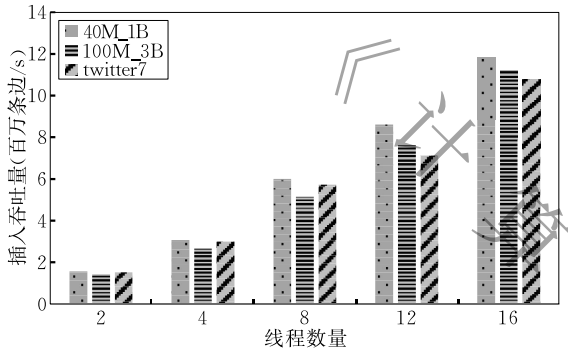


图 13　不同线程数目(2、4、8、12 和 16)LMSA 的插入吞吐量(百万条边/s)

我们在 4.2 节对 LMSA 与 AL、CSR 和 EL 等静态图数据结构进行对比，结果验证 LMSA 具有良好稳定性. 在 4.3 节对比 DRAM 与 NVM 环境下 LMSA 的性能，结果显示 NVM 环境下 LMSA 性能降低 1.7 倍，说明 DCPMM 的访问性能劣于 DRAM. 通过与 Stinger 和 GraphTinker 的插入与删除性能对比，LMSA 的插入性能相较 Stinger 提升 4.3～12.6 倍，相较 GraphTinker 提升 1.4～4.35 倍，删除性能相较 Stinger 提升 5.7～20.1 倍，相较 GraphTinker 提升 1.4～4.58 倍，说明 LMSA 的高效性. 在 4.4 节我们验证自适应 LMSA 在不同基础数组大小下性能变化，这种读写权衡的思想有助于进一步提升 LMSA 性能与提高空间利用率. 在 4.5 节，我们验证 C-LMSA 的空间压缩率会随着顶点度数的增加逐渐上升，它为 LMSA 的空间利用率提升提供思路. 4.6 节我们发现随着线程数量的增加，LMSA 插入吞吐量线性增长，结果验证 LMSA 具有良好扩展性.

5　总　结

本文介绍了在 NVM 环境下高性能动态图处理数据结构 LMSA，通过在对数时间内完成读写操作和无日志的一致性保障方案，它能够有效提升动态图数据的查询速度并能最大程度减少 NVM 写次数. LMSA 使用层级有序数组存储图顶点信息，这将减少传统静态图数据结构(AL 等)使用额外指针带来的空间消耗，从而提供良好的缓存局部性. 通过在配置 DCPMM 机器上的实验表明，LMSA 优于最新的动态图数据结构 Stinger 和 GraphTinker，是 Stinger 插入与删除操作吞吐量 4.3～12.6 倍与 5.7～20.1 倍，是 GraphTinker 插入与删除操作吞吐量 1.4～4.35 倍与 1.4～4.58 倍. 同时 LMSA 解决传统静态图数据结构在插入或删除或查询等性能急剧下降的问题. 本文提出的自适应基础数组大小 LMSA 通过对空间开销与查询延迟的权衡，可以进一步提升 LMSA 性能. 为了更进一步提高 LMSA 空间利用率，本文使用 Elias-Fano 编码方式实现 C-LMSA，实验表明在平均降低 20%更新吞吐量情况下它能够将 LMSA 层级数组位数压缩到原来的 9%～42%.

参　考　文　献

[1] Shu Ji-Wu, Lu You-You, Zhang Jia-Cheng, et al. Research progress on non-volatile memory based storage system. Science & Technology Review, 2016, 34(14): 86-94 (in Chinese)
(舒继武，陆游游，张佳程等. 基于非易失性存储器的存储系统技术研究进展. 科技导报，2016，34(14)：86-94)

[2] Zhang Hong-Bin, Fan Jie, Shu Ji-Wu, et al. Summary of storage system and technology based on phase change memory. Journal of Computer Research and Development, 2014, 51(8): 1647-1662(in Chinese)
(张鸿斌，范捷，舒继武等. 基于相变存储器的存储系统与技术综述. 计算机研究与发展，2014，51(8)：1647-1662)

[3] Yang Jian, 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 (FAST). Santa Clara, USA, 2020: 169-182

[4] Zuo Pengfei, Hua Yu, Wu Jie. Write-optimized and high-performance hashing index scheme for persistent memory//

Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI). CA, USA, 2018: 461-476

[5] Jeong H, Tombor B, Albert R, et al. The large-scale organization of metabolic networks. Nature, 2000, 407: 651-654

[6] Raman A, Tyson G, Sastry N. Facebook (A) Live? Are live social broadcasts really broadcasts?//Proceedings of the 2018 World Wide Web Conference (WWW). Lyon, France, 2018: 1491-1500

[7] Sengupta N, Bagchi A, Ramanath M, et al. ARROW: Approximating reachability using random walks over web-scale graphs//Proceedings of the 35th International Conference on Data Engineering (ICDE). Macau SAR, China, 2019: 470-481

[8] Jaiyeoba W, Skadron K. GraphTinker: A high performance data structure for dynamic graph processing//Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS). Rio de Janeiro, Brazil, 2019: 1030-1041

[9] Wang Xiong, Dong Yi-Hong, Shi Wei-Jie, Pan Jian-Fei. Progress and challenges of graph summarization techniques. Journal of Computer Research and Development, 2019, 56(6): 1338-1355(in Chinese)
(王雄, 董一鸿, 施炜杰, 潘剑飞. 图概要技术研究进展. 计算机研究与发展, 2019, 56(6): 1338-1355)

[10] Wu An-Biao, Yuan Ye, Qiao Bai-You, et al. The infuluence maximization problem based on large-scale temporal graph. Chinese Journal of Computers, 2019, 42(12): 2647-2664(in Chinese)
(吴安彪, 袁野, 乔百友等. 大规模时序图影响力最大化的算法研究. 计算机学报, 2019, 42(12): 2647-2664)

[11] Xu Jia, Zhang Qian-Zhen, Zhao Xiang, et al. Survey on dynamic graph pattern matching technologies. Journal of Software, 2018, 29(3): 663-688(in Chinese)
(许嘉, 张千桢, 赵翔等. 动态图模式匹配技术综述. 软件学报, 2018, 29(3): 663-688)

[12] Zheng Da, Mhembere D, Burns R C, et al. FlashGraph: Processing billion-node graphs on an array of commodity SSDs//Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST). Santa Clara, USA, 2015: 45-58

[13] Macko P, Marathe V J, Margo D W, et al. LLAMA: Efficient graph analytics using large multiversioned arrays//Proceedings of the 31st International Conference on Data Engineering (ICDE). Seoul, Korea, 2015: 363-374

[14] Ediger D, McColl R, Riedy E J, et al. Stinger: High performance data structure for streaming graphs//Proceedings of the 2012 IEEE High Performance Extreme Computing Conference (HPEC). MA, USA, 2012: 1-5

[15] Cheng R, Hong Ji, Kyrola A, et al. Kineograph: Taking the pulse of a fast-changing and connected world//Proceedings of the 2012 European Conference on Computer Systems (EuroSys). Pragu, Czech Republic, 2012: 85-98

[16] Ju Xiaoen, Williams D, Jamjoom H, et al. Version traveler: Fast and memory-efficient version switching in graph processing systems//Proceedings of the 2016 USENIX Annual Technical Conference (ATC). Santa Clara, USA, 2016: 523-536

[17] Feng Guoyao, Meng Xiao, Ammar K. DISTINGER: A distributed graph data structure for massive dynamic graph processing//Proceedings of the 2015 IEEE International Conference on Big Data (BigData). Santa Clara, USA, 2015: 1814-1822

[18] Green O, Bader D A. cuSTINGER: Supporting dynamic graph algorithms for GPUs//Proceedings of the 2016 IEEE Conference on High Performance Extreme Computing (HPEC). Waltham, USA, 2016: 1-6

[19] Kyrola A, Blelloch G E, Guestrin C. GraphChi: Large-scale graph computation on just a PC//Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Hollywood, USA, 2012: 31-46

[20] Dhulipala L, Blelloch G E, Shun Julian. Low-latency graph streaming using compressed purely-functional trees//Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Phoenix, Arizona, USA, 2019: 918-934

[21] Xu Jian, Swanson S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories//Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST). Santa Clara, USA, 2016: 323-338

[22] Yang Fan, Lu Youyou, Chen Youmin, et al. No compromises: Secure NVM with crash consistency, write-efficiency and high performance//Proceedings of the 56th Annual Design Automation Conference (DAC). Las Vegas, USA, 2019: 31:1-31:6

[23] Zhou Ping, Zhao Bo, Yang Jun, et al. A durable and energy efficient main memory using phase change memory technology //Proceedings of the 36th International Symposium on Computer Architecture (ISCA). Austin, USA, 2009: 14-23

[24] Chen Shimin, Jin Qin. Persistent B+-trees in non-volatile main memory//Proceedings of the 41st International Conference on Very Large Data Bases (VLDB). Hawaii, USA, 2015: 786-797

[25] Oukid I, Lasperas J, Nica A, et al. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory//Proceedings of the International Conference on Management of Data (SIGMOD). San Francisco, USA, 2016: 371-386

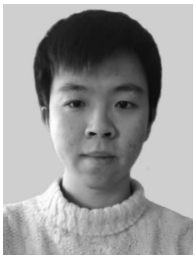
[26] Lim S, Lu Zaixin, Ren Bin, et al. Enforcing crash consistency of evolving network analytics in non-volatile main memory systems//Proceedings of the 15th International Conference on Parallel Computing Technologies (PACT). Almaty, Kazakhstan, 2019: 124-137

[27] Vigna S. Broadword implementation of rank/select queries//Proceedings of the 7th International Conference on Experimental Algorithms (WEA). Massachusetts, USA, 2008: 154-168

[28] Blelloch G E, Fineman J T, Gibbons P B, et al. Sorting with asymmetric read and write costs//Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). Portland, USA, 2015: 1-12

[29] Sengupta D, Song S L, Agarwal K, et al. GraphReduce: Processing large-scale graphs on accelerator-based systems//Proceedings of the Supercomputing 2015 (SC). Austin, USA, 2015: 28:1-28:12

[30] Murphy R C, Wheeler K B, Barrett B W, et al. Introducing the graph 500//Proceedings of the Cray User's Group (CUG). Edinburgh, UK, 2010: 45-74



ZHU He, M. S. candidate. His research interests include NVM-based data structure, high performance computing.

HUA Qiang-Sheng, Ph. D., professor. His research interests include parallel and distributed computing theory and algorithms.

JIN Hai, Ph. D., professor. His research interests include computer architecture, computing system virtualization, cluster computing and cloud computing, network security, peer-to-peer computing, network storage and parallel I/O.

LIAO Xiao-Fei, Ph. D., professor. His research interests include in-memory computing, runtime system, graph computing.

Background

Graphs have been widely used to represent information in different fields. Whether it is the structure of biological networks, the inter-entity relationship of social networks, or the links between Web pages in Web graphs, the graph representation method can be used for the storage and processing of relevant data. At the same time, the amount of information in these fields is growing and changing rapidly. Unfortunately, efficient graph processing for dynamic graphs presents many challenges. The first major challenge is the unpredictable edge update patterns in dynamic graphs, which will lead to poor graph update throughput. The second major challenge is how to design data structures that have good scalability. The third major challenge is that with the increase in the amount of graph data, the traditional DRAM memory hierarchy may not be able to meet the memory storage and computing requirements under massive graph data.

The existing dynamic graph data structures mainly include Stinger, GraphTinker, etc. Stinger implemented the dynamic graph processing framework based on the shared memory data structure of the adjacency list. The neighbor nodes in Stinger will be divided into pre-selected contiguous blocks, which form a linked list. All blocks are the same size except for the last block in each linked list. GraphTinker improves dynamic graph processing from the perspective of batch throughput and scalability. It uses a new tree hashing scheme to reduce the detection distance to improve edge update performance, but this approach leads to additional hashing overheads and hashing conflicts. At the same time, it uses a kind of mixed granularity compression edge data form to store the information of adjacent vertices, which can bring the improvement of space utilization.

This paper uses NVM to design a dynamic graph data structure LMSA that trades off read and write optimization. It can find the edge to be queried in $O(\log^2 d_{\max})$, and complete each insertion operation with $O(\log d_{\max})$ write times. Based on the idea of LMSA, the LMSA with adaptive basic array size and the C-LMSA with space compression are proposed, which further increases the space utilization and reduces the read and write operations on NVM. We use the Intel Optane DC Persistent Memory Module machine to conduct experimental evaluations of LMSA query, insert, delete, update and space overheads. Comparing with the latest dynamic graph data structure Stinger and GraphTinker, we verify the high efficiency of LMSA.

This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1003203 and the National Natural Science Foundation of China under Grant Nos. 61972447, 61832006.