软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(3):832−848 [doi: 10.13328/j.cnki.jos.006456]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向非易失内存的异构索引*

刘睿诚[1,2], 张俊晨[1,2], 罗永平[1,2], 金培权[1,2]

[1](中国科学技术大学 计算机科学与技术学院, 安徽 合肥 230026)

[2](中国科学院电磁空间信息重点实验室(中国科学技术大学), 安徽 合肥 230026)

通信作者: 金培权, E-mail: jpq@ustc.edu.cn

摘 要: 非易失内存(non-volatile memory, NVM)为数据存储与管理带来新的机遇, 但同时也要求已有的索引结构针对NVM的特性进行重新设计. 围绕NVM的存取特性, 重点研究了树形索引在NVM上的访问、持久化、范围查询等操作的性能优化, 并提出了一种上下两层结构的异构索引 HART. 该索引结合了 B+树与 Radix 树的特点, 同时利用了Radix结点搜索快以及B+树范围查询性能好的优点. 对整体架构进行了精心设计, 改进了Radix树的路径压缩策略, 设计了NVM写友好的结点结构, 并将Radix树叶结点集中存储和链接. 同时在仿真NVM设备以及傲腾真实NVM平台上进行了实验, 对比了HART的不同衍生变种的性能, 并与多个NVM索引进行了对比. 结果表明, HART的写性能和点查询性能优于现有的类B+树索引, 范围查询性能优于基于Radix的WOART索引, 具有较好的综合性能.

关键词: 非易失内存; 索引; 两层结构; 读写优化; ART树
中图法分类号: TP311

中文引用格式: 刘睿诚, 张俊晨, 罗永平, 金培权. 面向非易失内存的异构索引. 软件学报, 2022, 33(3): 832–848. http://www.jos.org.cn/1000-9825/6456.htm

英文引用格式: Liu RC, Zhang JC, Luo YP, Jin PQ. Heterogeneous Index for Non-Volatile Memory. Ruan Jian Xue Bao/ Journal of Software, 2022, 33(3): 832–848 (in Chinese). http://www.jos.org.cn/1000-9825/6456.htm

Heterogeneous Index for Non-volatile Memory

LIU Rui-Cheng[1,2], ZHANG Jun-Chen[1,2], LUO Yong-Ping[1,2], JIN Pei-Quan[1,2]

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

[2](Key Laboratory of Electromagnetic Space Information (University of Science and Technology of China), Chinese Academy of Sciences, Hefei 230026, China)

Abstract: Non-volatile memory (NVM) introduces new opportunities to data storage and management, but it also requires existing indices to be revisited according to the properties of NVM. This study, based on the access characteristics of NVM, focuses on the performance optimization of the access, persistence, range query, and other operations of tree indexes on NVM. A two-layer heterogeneous index called HART is presented. HART takes advantage of the high range query performance of the B+-tree and the fast node search speed of the Radix tree. The index structure is redesigned and the strategy of path compression for the Radix tree is improved. Moreover, a write-efficient node is proposed for NVM and the Radix leaf nodes together with a link are stored. The experiments are conducted on both simulated NVM and real Intel Optane persistent memory modules and different variants of HART are compared to several existing NVM indices. The results show that HART achieves better performance for point queries and writes than existing B+tree-like indices. In addition, it outperforms the existing Radix-tree-based WOART index in range query performance. As a result, HART can deliver high overall performance.

---

* 基金项目: 国家自然科学基金(62072419)
刘睿诚和张俊晨在本文工作中有同等贡献.
本文由"数据库系统新型技术"专题特约编辑李国良教授、于戈教授、杨俊教授和范举教授推荐.
收稿时间: 2021-06-30; 修改时间: 2021-07-31; 采用时间: 2021-09-13; jos 在线出版时间: 2021-10-21

Key words: non-volatile memory; index; two-layer structure; read/write optimization; ART tree

非易失内存(non-volatile memory, NVM)是近年来继闪存(Flash memory)后出现的新型存储介质. 它不仅和闪存一样具有掉电数据不丢失的优点, 还支持像 DRAM 一样按字节存取. 同时, 非易失内存的存储密度一般高于 DRAM, 因此可以提供比 DRAM 更高的容量, 解决当前 DRAM 密度低容量小的问题. 目前, 学术界和工业界针对不同类型的非易失内存开展了大量研究, 包括自旋转移力矩存储器(shared transistor technology random access memory)[1]、阻变式存储器(resistive random access memory)[2]、磁性存储器(magnetoresistive random access memory)[3]以及相变存储器(phase change memory)[4,5]等. 特别是 Intel 凭借与 Micron 共同研发的 3D XPoint 技术[6]提升了晶粒的存储密度, 于 2019 年 4 月推出了第一款商用的非易失内存——傲腾持久化内存(Intel optane DC persistent memory), 极大地促进了非易失内存相关的研究和应用.

在数据库系统领域, 新型非易失内存技术的引入降低了数据库系统部分数据的持久化代价以及存储在持久化介质上数据的访问代价, 这使得需要持久化的数据的快速存取成为可能. 数据库索引作为数据库提供高吞吐的必要组件, 比起在外存持久化, 更适合持久化在非易失内存上. 目前, 对于非易失内存上的索引已经有较多的工作, 包括 B+树[7−9]、跳表[10]、Radix 树[11−14]、日志结构合并树(log-structure merge tree, LSM-Tree)[15]、散列表[16−19]等方面的优化. B+树是传统关系数据库系统中普遍使用的索引结构, 因此非易失内存上的索引也大都以它为基础. 跳表、散列表和 LSM-Tree 则主要应用在键值(key-value)数据库系统中, 以优化点查询或者解决键值数据库中的高写吞吐需求为主要目标. 本文研究的异构索引是针对传统数据库应用领域,因此也以 B+树为基础, 暂不考虑目前键值数据库领域中的索引结构. 另一方面, 虽然 B+树具有平衡、多叉、有序、支持范围查询、高空间填充度等优点, 但它在查询时需要进行大量的键值对比, 会引入大量的缓存行读取操作, 影响查询性能. 因此, 本文考虑引入不需要键值对比操作即可快速定位记录地址的 Radix 树来优化 B+树的结构, 最终构建一种新的综合 Radix 树和 B+树优点以及 NVM 特性的异构索引.

总体而言, 本论文的主要贡献如下:

(1) 针对 Radix 树与 B+树在非易失内存上的性能表现差异, 提出了结合这两种索引特点的异构索引 HART(hybrid adaptive radix tree). HART 是以 ART (adaptive radix tree)作为内部搜索树、B+树作为子树与数据存储层的异构索引. 我们通过设计 NVM 写友好的结点, 将内存分布离散的叶子集中在一起存储, 利用局部性原理提升 NVM 与高速缓存的访问效率. 同时, 采用底层持久化链表使得 HART 获得与 B+树类型索引相当的范围查询性能. 此外, HART 通过改进 ART 树的路径压缩策略使得底层数据具备可恢复性, 同时增大了缓存行访问时的有效信息量.

(2) 我们同时在仿真 NVM 设备以及傲腾真实 NVM 平台上进行了实验, 对比了 HART 的不同衍生变种的性能, 并与多个 NVM 索引进行了对比. 结果表明: HART 的范围查询性能与 B+树类索引相近; 在完全均匀稀疏的分布下, 写性能与纯 NVM 索引相当, 但读性能则大幅优于其他索引.

本文第 1 节介绍国内外相关工作. 第 2 节讨论面向非易失内存的异构索引 HART 的详细设计. 第 3 节给出实验结果. 第 4 节总结全文工作并展望未来研究方向.

1 相关工作

如何利用好新型非易失内存的特性设计出新型索引, 达到快速查询、更新持久化介质上的数据, 一直是国内外索引研究的热点. 本节将按照时间顺序, 分别从 B+树与 Radix 树两方面介绍国内外的 NVM 索引研究现状.

1.1 基于B+树的NVM索引研究

目前, NVM 上的索引研究大致可分为两类: 面向纯 NVM 的索引结构以及面向 DRAM+NVM 混合内存的索引结构. 下面分别进行阐述.

在面向纯 NVM 的索引方面, 2011 年提出的 CDDS-Tree[20]是首个针对新型非易失内存技术下的 B+树索引,

© 中国科学院软件研究所 http://www.jos.org.cn

它对每组入口维护了两个时间戳, 在这基础上采用了多版本并发控制机制(multi-version concurrency ·control, MVCC), 以实现 NVM 中 B+树结点与键值的异位更新. 为应对指令重排序, 该索引使用了 mfence 和 clflush 微指令, 显式地控制高速缓存中数据的有序写回, 最终保证了数据更新的失败原则性. 但 CDDS-Tree 会维护结点内的有序键值对, 而这类排序操作会导致结点内键值对的移动, 再加上 MVCC 需要多次写入与回收, 导致索引的写放大较高.

2015 年提出的 wB+-Tree[7]考虑到 NVM 读写不一致的特性, 从 NVM 写优化出发, 以仅附加的方式在结点内添加新的数据, 不再维护物理存储有序, 期望以此减少写 NVM 的次数. 为了避免结点内线性搜索, 它在头部维护了存储逻辑顺序的排列与空间信息的位图. 由于有两个阶段的写, 因此在结点较大时, 需要日志来保证失败原子性.

2018 年提出的 FAST&FAIR 索引[21]作为 B+树更新、查询、分裂的改进策略, 提出了容忍读不一致的设计思想, 即崩溃后的系统在访问时不会返回错误结果, 而在后续更新时才修复不一致的失效指针. 因此, 它可以在不需要日志的情况下实现失败原子性.

另一些索引工作针对 DRAM+NVM 混合内存展开了研究. 由于 NVM 仍与 DRAM 存在读写性能上的差异, 索引也可以选择存放在混合内存架构中, 通过选择性持久树形索引的叶子结点(或数据), 使用 DRAM 缓存内部搜索树或热点数据, 加速内部搜索树的访问. 但由于需要在 DRAM 中存储索引和元数据, 因此 DRAM 需要足够大的空间, 总体部署成本也更为高昂. 此外, 在系统崩溃后, 恢复索引结构需要额外的时间扫描 NVM 中的数据, 并恢复 DRAM 里面的数据结构, 因此在单机系统上无法实现即刻恢复.

在面向 DRAM+NVM 混合内存的索引方面, 2016 年提出的 NV-tree[22]与 FPTree[23]都采用上层缓存友好的设计加速查询, 底层使用无序的叶子结点优化写性能, 且都免去了结构更新操作(structure modify operation, SMO)日志的维护, FPTree 还在叶子结点中使用 fingerprints 加速结点内无序键值对的查询, 以及硬件事务内存(hardware transactional memory, HTM)[24]处理中间结点并发. 2017 年提出的 HiKV[19]是面向 NVM 与 DRAM 混合架构下构建的索引, 它结合了散列索引与 B+树的特性, 通过散列索引减少 NVM 访问的同时, 借助 B+树提供更加丰富的键值操作, 如范围查询等. 2018 年提出的 ClfB-Tree[8]为减少 clflush 次数, 提出了结点大小为一个缓存行的 B+树设计, 使得只需要一次缓存行读或写就能访问或更新该结点, 结点内使用差分编码的方式压缩键与指针, 提升结点容量, 降低 NVM 写代价. 2019 年提出的 DPTree[25]采用了类似于 LSM-Tree[26]的架构, 在 DRAM 与 NVM 分别维护了两棵 B+树, 将写 DRAM 索引与持久化数据操作异步化, 降低了写响应时间, 缓冲带来的数据批量持久化也降低了 NVM 平均写代价. 2020 提出的 LightKV[27]以优化 LSM-Tree 的性能为目标, 提出了面向 DRAM+NVM 混合内存的 Radix 和散列相结合的索引结构. 2020 年提出的 LB+-Tree[28]提出了 entry moving 的设计, 在更新结点内数据与头部的同时, 将与头部处在同一缓存行的元组提前后移, 使得在后续插入新元组的过程中, 元组更新的位置大概率与头部位于同一个缓存行. 在最坏情况下, 平均每次插入代价由写 1.7 个缓存行减少到写 1.31 个缓存行, 减少了 NVM 写次数.

1.2 Radix树相关研究

Radix 树(如图 1 所示)作为 Trie 树的变种, 通过对键的每个分片来划分数据域, 常见于 IP 路由或内存管理. 作为确定性索引的一种, Radix 树在键分布确定时, 插入或更新顺序的不同也不会导致最终结构的不同. 与 B+树等不确定性索引相比, Radix 树不必多次把键与目标进行大小比较, 因此它能够高效地找到目标入口.

2017 年提出的 WOART(write optimized adaptive radix tree)[11]从 Radix 树的角度出发, 充分发挥 Radix 树结点访问无键值比较的特性, 因此查询性能优异, 在保证失败原子性上, 使用写时复制(copy on write, COW)保证数据的完整性, 在结点内添加深度信息, 以校验一致性并快速恢复. Radix 树由于在处理范围查询时需要回溯结点, 且所有键值存储离散, 这些特性都会导致大量的 NVM 读, 因此 Radix 树范围查询极差. 2021 年提出的 ROART(range-query optimized adaptive radix tree)[12]就从优化范围查询的角度出发, 提出了叶结点合并技术, 将 Radix 树中零散的叶子入口整合到一个叶子数组中, 减少回溯层数的同时, 利用局部性原理提升遍历速度. 目前, 操作系统的地址映射只有效利用了低端的 48 位, ROART 在高 16 位存放了 fingerprints 或键的局部

© 中国科学院软件研究所 http://www.jos.org.cn

分片, 在遍历时避免了大量不必要的 NVM 访问.

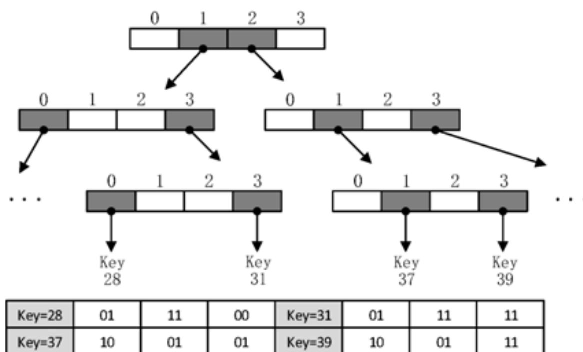


图 1 Radix 树结构

ART 在原有的结点基础上额外设计了 3 种不同扇出的结点 Node4, Node16, Node48, 扇出越小, 结点所占用的空间也越小, 以此来优化 Radix 树对稀疏分布下的空间代价. 各个结点的数据结构如图 2 所示. ART 中结点命名为 NodeX, 一个结点的分片为一个字节, 而 X 就代表了该结点的最大扇出. 在 Node4 中, ART 维护了一个存放键的数组与存放指向下层结点指针的数组, 该存放键的数组的偏移与存放指针的数组的偏移对应. 在 Node4 中, 键都是被有序地排列在列表前端. 在 Node16 与 Node4 类似, 键与指针同样被分开存储在两个长度为 16 的列表内. 在扇出超过 16 后, 由于对结点内键的查询代价开始随入口对的增加而增大, 因此 Node48 维护了一个长度为 255、存放指针数组偏移量的数组, 在查询时, 以该结点的分片作为该数组的偏移量进行访问, 得到指针数组的偏移量, 找到对应的孩子指针或入口指针. 在 ART 中, Node4 是新建内部结点时的初始类型, 面向 2–4 个对象的存储; 在存储数量超过 4 后, Node4 会被 Node16 替代; 同理, Node48 是面向 17–48 个对象; 而 Node256 作为 Radix 树的原本结点, 可容纳 49–255 个对象. 在中间结点包含的键数量增加时, 会升级到对应的结点类型; 减少时可以降级, 也可以延迟回收的方式回收, 避免 4 种类型反复切换.

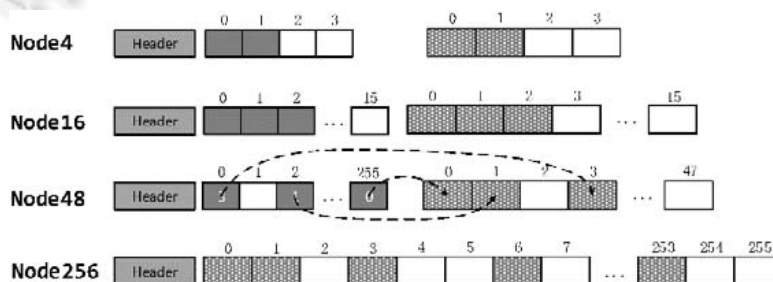


图 2 ART 索引结构

2 面向 NVM 的异构索引结构

本文针对非易失内存的存储特性, 考虑到 B+树与 Radix 树在不同场景下的优劣, 设计了一种面向 NVM 的异构索引 HART. HART 是以 ART 作为内部搜索树、B+树作为子树与数据存储层的异构索引. Radix 树的路径压缩策略的引入, 使得整棵树有从上向下的路径分裂, 也有 B+树从下向上级联的结点分裂. 因此, 我们对 Radix 树的压缩策略进行了重新设计. 同时, 针对 NVM 写优化, 我们重新设计了 B+树内部结点, 使其在不依赖日志情况下保证失败原子性.

2.1 研究动机

在树形索引中, 无论是 B+树还是 Radix 树, 都是借助内部搜索树来找到存储键值对或其集合的结点的入

© 中国科学院软件研究所 http://www.jos.org.cn

口, 再做出进一步的查询、更新、插入、范围查询等操作的. 因此, 索引的优化工作除了维护特性而需要的结构修改操作(structure modification operations, SMO)优化外, 还有内部搜索树的查询优化与结点内的读写操作优化等. 接下来以 B+树为例, 讨论存储介质的变化对优化目标带来的影响以及在这变化中产生的机遇.

2.1.1 树形索引层高变化

在传统磁盘B+树里, 由于磁盘IO代价较高, 且操作系统以页为单位来管理控制内存空间的换入换出, 因此磁盘 B+树的结点扇出一般较高.

以 MySQL 的 InnoDB 存储引擎为例, 表空间的页是 InnoDB 基础的存储单元, 每个页默认容量是 16 KB, 中间结点的一组键与指针对大小是 14 字节, 因此每个存储单元能存放至少 1 K 组键值对. 假设叶结点中键值对大小是 1 KB, 那么 3 层高的 B+树就能满足索引至少 16 M 项的数据. 也即对于千万级的数据量, B+树的层高也只需 3 层就能建立索引, 而 InnoDB 中表空间的页大小是可以进行调整、以适应更大规模的数据集的, 因此很少会出现超过 4 层的案例.

但是, 当我们把 B+树迁移到非易失内存上时, 由于 NVM 的延迟与吞吐相比于磁盘更加接近于 DRAM, 访问代价中占比最高的外存 IO 变成了 NVM 的访问与高速缓存的刷回, 总代价与访问路径上的结点数量、每个结点内的访问代价相关. 例如, 在 Intel 傲腾持久性内存上, 相关研究发现, 由于 XBuffer 等内部设计, NVM 上数据的存取粒度是 256 字节或其整数倍时, 性能才会与顺序访问时的带宽相当[29,30]. 此外, 在 B+树的中间结点内的遍历, 是通过二分搜索来找到下一页指针的, 过大的中间结点使得在结点内查询时, 也会面临着更多的高速缓存的换入换出.

因此, 为优化 NVM 上的读写与高速缓存的换入与写回, 存储在 NVM 上结点的大小应当被设置为 NVM 的最优存取粒度或其整数倍, 但会小于磁盘 B+树的结点大小. 如果暂不考虑失败原子性的保证措施与结点头部所占的空间, 在结点大小为 512 B 时, 索引结点最大扇出为 32, 对于同样 16 M 项的数据, B+树的内部搜索树至少会有 7 层高. 而在 Radix 树中, 对于 8 字节长的键最坏情况下也会有长度为 8 的搜索路径.

因此, 由于存储介质的变化, 索引的搜索路径会变得更长, 而这为基于不同模型的异构索引带来了应用空间.

2.1.2 Radix 树与 B+树的优势区间

由于 B+树的内部结点在查询时需要大量键值对比, 即便结点内键值对有序存储, 每个结点上的也需要多次载入缓存行才能缩小查询范围, 找到查询键对应的入口. 而对于确定性索引 Radix 树而言, 结点内部结构由插入的键确定, 因此不需要对搜索键进行大小比较而多次载入缓存行. 因此, 在内部搜索树上查找特定值上面, Radix 树理论上表现更优.

但是, Radix 树结点的空间利用率比 B+树低, 且键值对入口分布零散, 在存取的粒度小于 256 字节时, 零碎的叶子设计对 NVM 读写不友好. 此外, Radix 树的键值对入口的物理分布也过于零散, 这对范围查询的场景也极不友好, 需要回溯大量的中间结点, 而这会带来大量的 NVM 访问. 最后, Radix 树也因为该特性, 难以划分出可不持久部分, 无法使用选择性持久化策略, 或直接在混合内存架构下部署.

另一方面, B+树作为一棵平衡树, 其叶结点可以把键值对按照统一的结点类型集中存储在一起, 方便内存空间的申请与释放, 也因为局部性原理, 更符合高速缓存的预取策略使用场景, 提升小范围的范围查询性能. 在 B+树叶结点中的兄弟指针, 对提升较大范围的遍历的效率也有不错的效果. 因此在数据存储的组织结构上, B+树相比 Radix 树更易管理、访问更加高效.

但在非易失内存上部署 B+树和 Radix 树两种索引时, 还需要保证结构的失败原子性. 如果在传统内存索引结构的基础上添加这些保证措施, 会产生额外的代价. 对于结构修改操作, B+树上有结点的分裂与合并, ART 有路径分裂与插入新结点. 前者由于对于结点而言是原位更新, 需要使用 COW 或额外日志的方式保证, 或是按照 FAST&FAIR 中提到的容忍读不一致修改算法; 后者由于都是异位更新, 只需要像 WOART[11]保证最后更改指针引用, 就能保证失败原子性. 因此, 在失败原子性保证上, 即便采取了 wB+-tree 的结点内不维护物理存储顺序的策略, B+树类的索引也比 ART 面临更大的挑战.

2.2 HART整体结构

将非易失内存引入至数据库系统的索引部署中, 无疑可以降低持久化代价, 但目前学术界大多把目光放在 B+树上, 包括不使用写时复制技术或日志进行更新、在缓存行内进行异位更新、使用微日志、选择性持久化等. 也有学者指出: B+树的中间结点结构的查询表现不及 Radix 树[11], 结点分裂与排序使得持久化代价高昂, 优化困难, 因此在 NVM 上的索引, 把 Radix 树作为架构是更好的选择. 但 Radix 树也因为其低效的回溯遍历方式, 使得范围查询性能难以与其他索引结构相比, 因此也有学者提出把数据存储层的键值对合并成一个结点, 以优化范围查询性能.

前面我们讨论了 ART 与 B+树展现出的不同的优势与劣势区间, 可以使用 B+树解决数据存储层的分裂与平衡问题, 加上持久在 NVM 上的树形索引的层高会比持久化在机械硬盘或固态硬盘上的高, 适合分层使用异构模型, 再考虑到 NVM 的容量比 DRAM 更大, 可以容忍 Radix 树一定程度上的空间放大, 我们就提出基于 ART 与 B+树两种树形索引的异构索引 HART.

如图 3 所示, HART 的整体架构可分为两层: 上层的 ART 结构和下层的 B+树结构. 由于 ART 的内部搜索树表现出优异的单值查询性能, 因此我们延用其大致结构, 把 ART 作为异构索引 HART 的内部搜索树的上层. 与使用 B+树作为上层结构相比, 采用 ART 结构可以有效降低上层索引的键值查找代价. B+树虽然在失败原子性保证上会面临更多的挑战与困难, 但仍适合作为数据存储层, 以降低相近键的数据在 NVM 上的离散程度, 提升小范围查询性能, 而 B+树底层的链表也可以改善相对较大范围的查询性能. 因此, 把 B+树作为持久化链表上的存储层.

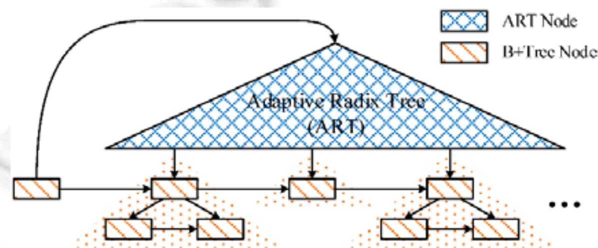


图 3 HART 索引整体结构

HART 的结构包含了下面两个主要的新设计.

(1) 使用 NVNode 置换 ART 叶子

在图 3 所示的 HART 整体架构中, 由于 Radix 树不是平衡树, 由于路径压缩的引入, 其叶子分布在路径上, 不能直接套用原本的 ART 的架构.

但若不对路径进行压缩, 又会在没有分叉的路径上产生许多的无用结点, 造成空间浪费, 也会增多 NVM 的访问次数, 影响查询性能.

因此, 我们修改了 ART 的结构, ART 中不再有“叶子”的概念, 而是以 NVNode (non-volatile node)作为替代, 在插入新值但找不到对应的 NVNode 时, 就申请新的空间创建一个集中存放键值的 NVNode.

如图 4 所示, NVNode 作为该 B+树的叶结点, 同时也可以是这棵子树的根结点. ART 与各子树之间不存在中间元数据结点或入口结点, 上层指针直接指向 NVNode, 以减少索引层高与 NVM 的访问次数. 为了优化空间使用率, NVNode 与 ART 结点之间也会采用路径压缩. 图 4 中所示的 NVNode 等结点的结构将在第 2.3 节中详细进行阐述.

(2) 持久化链表与选择性持久化

在持久性内存上管理数据, 必须指定一个数据恢复的入口, 以方便数据的管理与访问. 由于 ART 中不在存储任何键值对, 因此我们采用选择性持久化策略, ART 的结点虽然是存放在 NVM 上, 但不对其结点的更新保证是原子的, 以降低失败原子性保证的代价.

© 中国科学院软件研究所 http://www.jos.org.cn

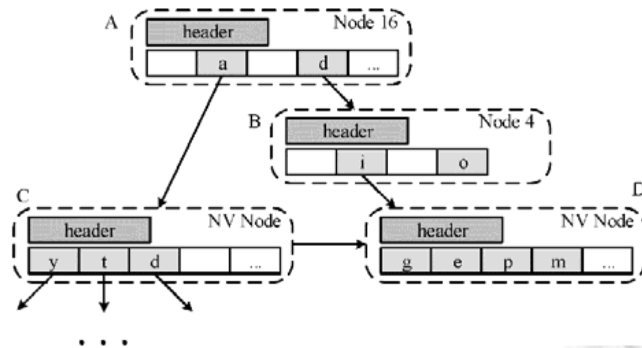


图 4 HART 结点结构(局部)

同时，下方的数据存储层的入口被链表链接起来，一方面针对范围查询进行了优化，另一方面则是为了保证索引的可恢复性.

由于上层的结点不保证持久化是及时、有效的，因此在掉电或故障后，上层结点可能会出现不一致甚至数据丢失的情况，因此恢复时不再以从 ART 根结点开始下降的顺序，而是直接从持久化入口遍历数据层，对上层 ART 进行校验和修复.

2.3 结点结构

本节将阐述 HART 涉及的 5 类结点的数据结构与关联较密切的算法，包含 Node4, Node16, Node48, Node256 和 NVNode，以及采用的结点分裂算法和路径压缩策略. 如图 5 所示，HART 的内部搜索树结点延用了 WOART 的 4 种类型的结点 Node4, Node16, Node48 以及 Node256，并对其做了一定程度上的改进，底层结点使用了 NVNode.

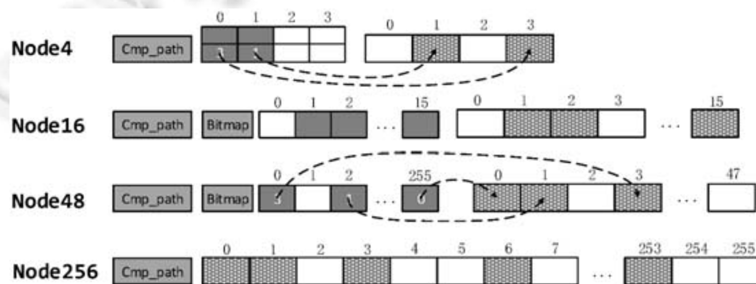


图 5 HART 内部搜索树结点结构

(1) Node4

为减少结点大小，ART 推出了容量上限为 4 的结点，它有序存储了键分片与对应的指针，但维护结点内数据存储上的有序不利于崩溃一致性的保证，需要以类似日志的某种形式来记录变更.

因此，WOART 将存放键的数组变更为两个位置关系对应的数组：一个存放键分片，一个存放键对应的指针在指针数组中的位置. 有了显式表示位置关系的中间层，就可以不对位置进行移动，以腾出插入的空间，也不必对结点内的键值对进行排序，而是以仅附加的形式在可用空间插入新值.

由于两个记录键与顺序的数组的大小之和正好为 8 字节，因此写入数据时，这两个数组可以保证原位原子更新，即在更新时将它们作为结点数据的有效标识位最后一起更新，从而满足崩溃后数据的一致性.

(2) Node16

Node16 的容量上限为 16，面向存储 5−16 个键的场景. 在 ART 中，Node16 也维护了两个位置关系对应的有序数组，分别存放键分片与指针. 由于数组容量变大，无法实现 8 字节原子更新，WOART 因此没有使用中间层表示指针数组的位置信息，直接延用 ART 的隐式表示方式. 但与 Node4 的改进相似，没有维护数组的有

序, 以只附加的形式在可用空间插入新值.

为保证失败原子性, WOART 使用了位图来作为键分片数组的标识位, 表示分片数组的有效空间使用. 由于 Node16 的位图小于 8 字节, 因此可以原子更新, 在最后持久化可以保证失败原子性.

(3) Node48

Node48 的容量上限为 48, 面向存储 17–48 个键的场景. 在 ART 中, 维护一个容量为 256(键分片所能代表的最大范围)的数组和容量为 48 的指针数组. 容量为 256 的数组以键分片作为索引, 存储指针数组的偏移量.

WOART 只对更新流程做出了持久化改进, 即最后更新结点内的索引数组. 但我们在后续对 WOART 的性能分析中发现, Node48 无论在切换其他类型的结点还是在插入新值时都会扫描空间率至多只有 18.75%的索引数组, 以确认有效空间使用, 再访问后面的指针数组.

因此, 我们在 Node48 的头部中添加了一个位图, 表示索引数组的空间使用状态, 来优化 Node48 的插入性能. 虽然位图的大小正好为 8 字节, 但由于索引数组的更新是依据键分片进行的, 不会产生冲突, 已经可以保证失败原子性. 因此, 为减少持久化的步骤, 不选择增加额外的内存屏障与高速缓存刷回, 以保证失败原子性代价不会额外增加.

(4) Node256

Node256 作为 Radix 树的原生结点类型, ART 与 WOART 均未做过多的修改. 由于该结点自身只包含一个以键分片索引的指针数组, 因此不会出现崩溃不一致的问题. 我们也将保持 Radix 树的基本结点, 除了锁位以外不做过多变动.

(5) NVNode

NVNode 作为 HART 下层 B+树子树的基本存储数据类型, 其结构如图 6 所示. NVNode 大小有 256 字节, 并以高速缓存行的大小 64 字节进行对齐. 在非易失内存上, 以 256 字节或其倍数为存取粒度进行存取操作时, 能够获得比不对齐时更高的带宽[29], 因此, 我们将结点大小设置为 256 的倍数.

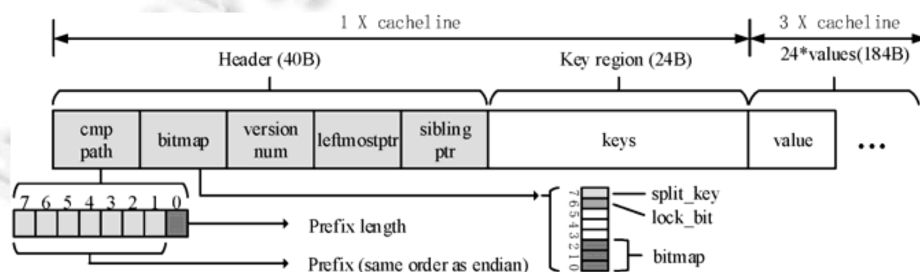


图 6　NVNode 结点结构

但是, 在结点的扇出超过 63, 也即对齐后结点大小超过 576 字节时, 小于或等于 8 字节的作为标志位的数据结构, 如位图等, 无法进行原子更新, 这样意味着需要建立额外的日志策略保证失败原子性, 或是建立内部索引的二级标志位, 进行额外一次的高速缓存行刷回.

而对于结点大小为 512 字节时, 一个缓存行无法同时容纳存储键分片的数组与结点头部, 这意味着在结点中存储的入口数超过 24 项后, 写操作也会额外多一次高速缓存刷回. 虽然该额外的刷回可以通过修改当前的结构来避免, 即不采用键分片与入口指针分离存储的策略, 而采取让键分片与入口指针存储在一个槽中, 保证在一个缓存行内一起刷回的方式, 但是由于内部数据结构对齐产生的内部碎片(padding), 此时扇出至多有 31, 与目前 25 的扇出相比, 结点花费额外 1 倍的空间, 让扇出只提升了 24%, 对降低子树树高难以起到较好的效果. 因此, NVNode 的结点大小被确定为 256 字节.

我们在 NVNode 的主体部分维护了两个数组: 一个是存储着键分片的键域, 一个是存放值或下一级的入口的数据域, 两个数组的下标相互对应. 其中, 键域由于存储的是键的 1 字节的键分片, 因此在集中存储的情况下不会出现结点内数据结构对齐产生的碎片, 结点从而会获得较高的扇出. 另一方面, 键域与结点头部在

同一缓存行, 因此在完成写操作后, 在更改头部标志位时可以在同一个缓存行里刷回, 减少了一次由于键值分离导致的额外一次缓存行的刷回.

在失败原子性保证上, NVNode 通过把结点头部的位图作为标志位, 位图的末 24 位代表着键域与数据域的有效空间使用情况, 首 8 位则作为键分片的分隔值, 第 9 位作为锁位进行并发控制.

NVNode 结点内插入/更新算法如算法 1 所示. 在此处, 键的 1 字节分片已经取出, 并在下降函数中, 已经通过路径压缩的验证. 为方便叙述, 下面我们直接以键来代指键的 1 字节分片. 在写入一个键值对时, 需要对是否更新做出判断, 因此, 算法首先对键域进行遍历, 若找到与搜索键相同的键, 则直接更新对应的值. 由于值为 8 字节, 可以直接原子更新, 不必考虑失败原子性保证. 在确认是新插入键值对后, 将在位图中找到可用的槽, 并分别对值与键进行更新. 由于值与键不在同一缓存行, 需要显式写回值所在的缓存行. 当数据写入完成后, 在最后更新作为标志位的位图, 并显式写回位图所在的缓存行. 由于键域和位图在同一缓存行, 并且键域的更新与位图的更新之间有内存写屏障, 因此可以省去一次缓存行写, 并同时保证失败原子性.

算法 1. NVNode 结点内插入/更新算法.

Input: *nvnode*, *key*, *value*.

1: **for** *slot* in *nvnode*.*bitmap* **do**
2:   **if** *nvnode*.*kr*.*keys*[*slot*]==*key* **then**
3:     *nvnode*.*records*[*slot*]←*value*;
4:     **return**;
5:   **end**
6: **end**
7: *slot*←find the *empty slot* in *nvnode*.*bitmap*;
8: *nvnode*.*records*[*slot*]←*value*;
9: *nvnode*.*keys*[*slot*]←*key*;
10: *clwb*(&*nvnode*.*records*[*slot*]);
11: *sfense*(·);
12: *set_bitmap*(*nvnode*.*bitmap*,*slot*);
13: *clwb*(&*nvnode*);
14: *sfense*(·);
15: **return**;

HART 采用了字节序的顺序存储方式, 也利于写入与验证压缩路径: 写入时直接把创建该结点的键写入该字段, 再把末尾 1 字节覆盖为有效长度; 验证时直接与搜索键相与, 非零的键分片就是与压缩路径不匹配的部分. 由于存储的不是路径压缩折叠的路径分片, 而是结点内所有键的共有前缀以及有效长度, 因此对于 NVNode 而言, 键域与前缀分片数组共同构成了这个叶结点内全部键的所有信息, 无需依赖访问其他结点.

2.4 点查询操作

在 HART 中, 为找到指定键是否被 HART 索引, 并返回对应的值, 只需要从根结点到叶结点逐层查询. 在上层 Radix 树中, 凡是未在结点内找到对应键分片的情况, 以及下降过程中结点有效前缀与搜索键不匹配时, 会直接返回未找到, 不必继续查询下去. 在下层 NVNode 中, 由于是以键大小比较的形式下推, 会下降到叶结点后才能返回结果. 虽然 HART 中凡是存放了键分片的结点都没有维护结点的有序性, 但所有结点的点查询不会载入超过 2 个缓存行.

下面对各个结点内部查询过程进行简要的阐述.

(1) Node4: 结点内无序, 但一个缓存行载入就可以把所有键分片载入高速缓存内, 线性搜索找到键分片, 再访问对应位置的指针.

(2) Node16: 结点内无序, 与 Node4 类似, 通过位图访问所有有效槽, 线性搜索找到键分片, 再访问对

© 中国科学院软件研究所 http://www.jos.org.cn

应位置的指针.

(3) Node48: 以键分片作为索引访问键分片数组, 获取指针数组偏移量, 再访问指针数组.

(4) Node256: 以键分片作为索引访问指针数组, 直接获得下一跳指针.

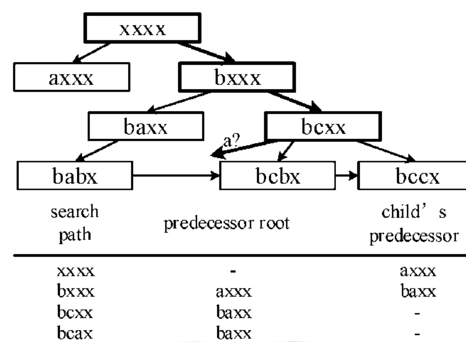
(5) NVNode: 叶结点与 Node16 相同. 中间结点会以线性搜索找到小于搜索键的最大键分片, 再访问对应位置的指针.

2.5 前驱结点查询

在进行范围查询时, 必须要找到一个叶结点作为开始结点, 遍历存储的键值, 返回一个有限的有序结果集合. 但这可能存在不给定一个开始键, 或开始键不在索引中的情况. 此时, 我们需要找到结点存储范围包含或相邻该开始键的结点, 即我们需要给定一个键, 找到覆盖范围最近的叶结点. 而在插入新中间结点、压缩路径分裂时, 可能会出现更新下层持久化链表的情况. 与 B+树结点分裂不同的是, 发生插入新结点、分裂出的新路径不一定会在当前结点之后. 在单向链表中找到触发该操作的结点的后继结点很容易, 但我们还需要一套算法找到其前驱结点.

出于以上范围查询与新结点插入的需要, 我们需要一个找到目标 NVNode 结点的同时, 获得前驱结点的搜索算法. 我们选择在从根结点下降的途中, 维护一个前驱根结点的指针变量, 以满足当前结点的前驱结点是这个前驱根结点的最右侧的中间结点或叶结点. 在下降到 NVNode 层时, 从前驱根结点开始查找这棵子树下最右侧的NVNode结点. 在不考虑并发的情况下, 该NVNode结点就是我们要找的目标位置的前驱结点, 就可以继续下一步的操作, 如范围查询、插入新结点等.

图 7 给出了一个前驱结点的查找流程例子. 图中结点将直接以 cmp_path 字段内容来直接表示该结点, 以方便直观地叙述. 同时键长为4, x代表无效的掩码. 假设我们要插入新值bcaa, 那么按照ART的查询路径, 将会按照图中加粗的路径下降; 但找不到键分片 a 时, 不依赖其他数据结构只能以回溯的方式进行查找. 但在HART中, 我们首先从根结点xxxx开始查询, 在xxxx结点内, 会依据键分片b找到bxxx的指针, 以及查找有效的前一位分片 a 与 axxx 的指针, 那么在搜索到 bxxx 结点前, 前驱根结点更新为 axxx. 同理, 下降到 bcxx 结点前, 前驱根结点更新为baxx. 在bcxx结点内查找a分片时, 未能找到a, 甚至未找到a的有效的前一位分片, 因此前驱根结点不更新, 继续下降. 由于此时未能在bcxx结点中找到a分片, 需要在下层链表上新建结点并插入, 那么前驱根结点 baxx 的最右的 NVNode 类型的结点 babx 就是我们要寻找的前驱结点.



search path	predecessor root	child’s predecessor
xxxx	-	axxx
bxxx	axxx	baxx
bcxx	baxx	-
bcax	baxx	-

图 7　键 bcax 的前驱结点查找示例

2.6 NVNode结点分裂

在HART中, 结构修改操作主要包含NVNode的结点分裂、新NVNode的插入以及被压缩路径的分裂. 当NVNode中存储的入口数达到了系统预设的容量阈值, 会触发结点分裂, 流程与B+树的结点分裂类似, 如图8所示, 分为新结点的分配与初始化、更新原结点、挂载新结点这3个步骤. 由于新结点在挂载前都不可见, 因此对恢复前后的线程而言只会暴露后两个步骤, 即一个中间状态.

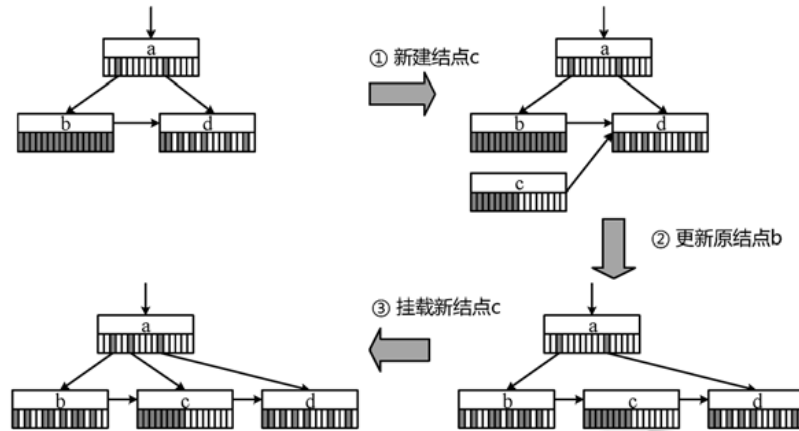


图 8　NVNode 结点分裂

(1) 新结点的分配与初始化

由于 NVNode 的结点内部没有维护键分片的有序性, 因此需要在这个无序的分片数组中找到中位数, 并依据中位数将入口分为两类. 我们采用了一个大顶堆和一个小顶堆来进行划分, 以避免完全排序. 大顶堆的键作为原结点残留的键, 小顶堆的键作为新结点被拷贝过去的键. 由于在 NVM 上访问代价比 DRAM 上高, 并且直接进行移位会导致无法保证失败原子性, 因此我们把键域的键分片拷贝进DRAM中的大顶堆和小顶堆中, 之后会不断交换大小顶堆的堆顶值, 直到大顶堆的堆顶小于或等于小顶堆的堆顶. 交换结束后, 键域就被大小顶堆平等地划分成了两部分.

随后, 按照小顶堆的键分片, 把原结点的对应的键分片、入口指针、以及结点头部中的键分片的分隔值、兄弟指针、压缩路径拷贝进新结点中, 更新新结点位图与左孩子指针, 并写入 NVM, 完成新结点的创建.

(2) 更新原结点

此时新结点尚未挂载, 因此虽然已经被持久化, 但并不会被访问到, 因此不会出现范围查询脏读的问题. 在新结点完成持久化后, 依次更新兄弟指针、键分片的分隔值与位图. 由于各个字段均小于等于 8 字节, 在有内存屏障的情况下可以保证有序持久化. 而对于只更新了兄弟指针未更新位图的情况, 虽然会出现冗余数据, 但不会出现数据丢失或不一致的情况. 最后, 在一次缓存行写回结点头部后, 原结点更新完毕.

(3) 挂载新结点

此时点查询无法直接访问到新结点, 而会访问到分裂前的原结点, 但通过头部中的键分片的分隔值, 可以确认是兄弟结点, 借助兄弟指针跳转访问到尚未挂载到父结点的新结点. 在原结点完成更新后, 就把新结点的指针插入到父结点中, 挂载新结点使得可以正常访问. 若没有父结点, 则新建根结点, 把两个结点的指针放入根结点中, 并更新新根结点头部. 完成新更结点的更新后, 最后再把上层 Radix 树中, 对原结点的指针原子更新为新根结点的指针. 至此, NVNode 的结点分裂结束.

2.7 路径分裂

当在 HART 插入新值, 但在上层 Radix 树的下降过程中, 发现结点前缀与新键不匹配时, 会触发路径分裂, 以展开被压缩的路径, 创建新的分支. 与 B+树结点分裂不同的是, 路径分裂是由上而下访问时触发的, 不会出现级联分裂的情况.

路径分裂可分为两个步骤: 新内部结点的分配与初始化、新内部结点的挂载. 图 9 显示了 HART 路径分裂的一个例子.

(1) 新内部结点的分配与初始化

在校验当前插入键与当前结点的压缩路径是否相符时, 我们通过搜索键与 cmp_path 字段执行与操作来实

现. 在插入验证的同时, 也可以获取共同前缀长度. 对于图 9 所示的例子而言, 在插入算法下降至结点 C 时, 我们会把新键 aabbcde 与结点 C 的压缩路径相与, 获得共同前缀 aabbc 与有效长度 5. 由于共有前缀有效长度 5 小于结点 C 的前缀有效长度 6, 那么就意味着需要进行路径分裂. 完成结点 D 的建立与持久化, 在创建新内部结点 B 时, 之前得到的新共同前缀 aabbc 与有效长度 5 将作为新内部结点 B 的压缩路径. 原分支 C 与新分支的结点 D 的指针也会被插入至结点 B 中. 最后, 待结点 B 的持久化结束, 就完成了新结点的分配与初始化.

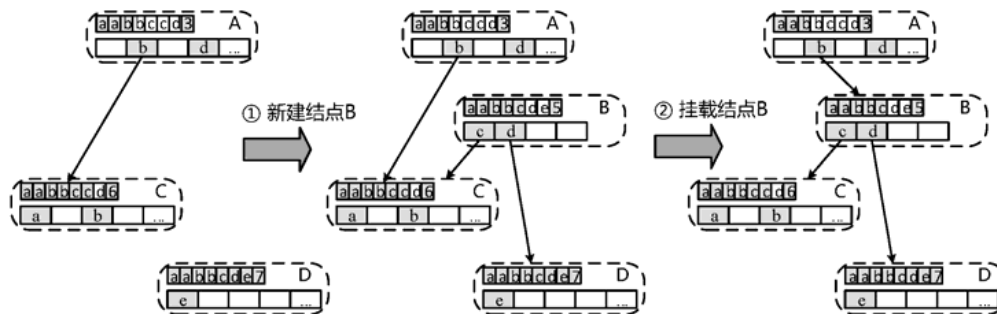


图 9 HART 路径分裂

(2) 新内部结点的挂载

由于结点 B 尚未挂载, 因此 BD 分支上的结点不会在点查询中被访问. 在指向结点 C 的指针原位更新为指向结点 B 的指针后, 被压缩的路径被展开, 新的分支就可以被访问. 由于指针长度为 8 字节, 因此本次原位更新是原子的, 只要使用内存屏障保证这个更新操作是最后执行的, 就能保证路径分裂的过程是原子的.

与 WOART 相比, 由于修改了路径压缩字段的存储, 因此不必修改结点 C 中的任何数据, 减少了一次 NVM 写.

3 实验与分析

3.1 实验设置

我们使用基于 XPoint 技术的 Intel 傲腾持久化内存搭建了实验平台, 并使用 Intel 提供的持久化内存开发工具包(persistent memory development kit, PMDK (https://pmem.io/pmdk/))的 libpmemobj 库来完成持久化内存的分配与管理. Intel 傲腾持久化内存支持 Memory 与 AppDirect 两种内存模式: 前者属于对传统易失性内存的兼容模式, 后者才是面向用户直接进行控制持久化控制的模式. 因此, 我们使用 Intel impctl 工具配置持久化内存为 AppDirect 模式, 并使用 ntctl 工具创建内存的命名空间, 以划分设备的逻辑分区. 最后设置设备的文件系统, 并进行挂载. 之后的操作与仿真实验环境配置类似, 配置的 DAX 文件系统使得用户能通过 load 和 store 指令以 64 字节的粒度进行对 NVM 访问, 并以 mmap 完成虚拟内存空间的映射, 绕过文件系统页高速缓存页, 避免了寻呼代价[31]. 表 1 给出了 Intel 傲腾持久化内存系统的详细参数.

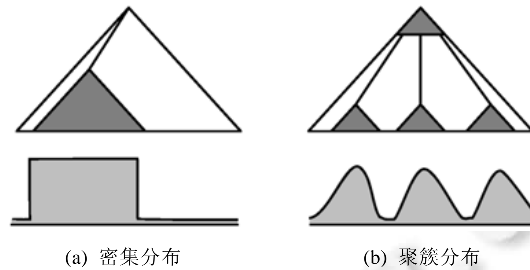
表 1 傲腾 NVM 真实环境参数设置

参数	配置
CPU	IntelXeon(R) Platinum 8276, DualSocket, 56 cores at 2.2 GHz, 32 KB iCache & 32 KB dCache
L2 Cache 大小	1 MB
L3 Cache 大小	38 MB (12 路)
DRAM 大小	384 GB (2 sockets×6 channels×32 GB)
NVM 大小	1 TB (2 sockets×4 channels×128 GB)
OS	Linux (kernel 5.8.7)
PMDKVersion	1.8

在数据集的选取上, 我们随机生成 8 字节的键与 8 字节的值. 所有实验默认的数据集包括 256 M 项记录, 整个数据集总大小为 4 GB.

© 中国科学院软件研究所 http://www.jos.org.cn

Radix 树由于是确定性索引, 索引结构由键的分布决定, 因此性能会因为键在数据集的分布的不同出现较大差异; 而上层架构采用 Radix 树的变种 ART 的 HART 也会受此影响, 因此实验中将构造两种同键分布的数据集, 对索引进行测试. 图 10 显示了两种键的分布. 每张图的上半部分所展示的是上层索引的大致结构, 其中, 阴影部分就是绝大多数的结点, 直线代表稀疏的路径或是被展开的压缩路径, 其余空白则表示整个索引范围内尚未申明使用的空间. 下半部分展示了键的密度分布, 越高则表示单位区域内键的个数越多.



(a) 密集分布 (b) 聚簇分布

图 10 两种不同键分布的数据集

在密集(dense)分布上, 所有键以连续自然数的形式密集集中在一起. 这导致键的头几个分片没有区分度, 因此路径压缩会出现在上层, 让访问跳过这些没有区分度的分片. 由于键以连续自然数的形式稠密分布, 因此此时 HART 的叶结点空间利用率较高.

在聚簇(clustered)分布上, 数据集被划分为多个簇, 各个簇之间不会重叠, 簇内以密集分布, 簇在整个数据集上均匀分布. 聚簇分布让键在整个数据集上呈现局部密集、局部稀疏的非均匀形式. 在聚簇分布的数据集上, 由于数据整体上看在整个数据集上均有分布, 而局部的分布也更加密集, 因此键的各个分片均会有区分度, 且末尾的区分度较高, 因此路径压缩只会出现在中间几层.

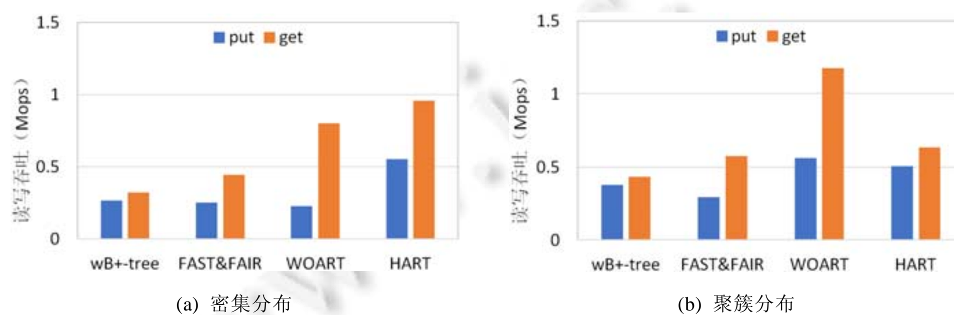
对于所有类型的分布, 在写入或查询时, 均完全打乱顺序, 通过单线程随机地进行插入与访问.

3.2 实验结果

我们把 HART 和 FAST&FAIR、wB+-tree、WOART 部署在真实的傲腾持久化内存上进行性能对比实验, 采用纯 NVM 内存架构. 实验对比主要考虑了读写吞吐、点查询性能、范围查询性能以及索引的空间代价. 由于 HART 索引的设计动机是综合 Radix 树和 B+树的优点, 因此我们预计, HART 索引的查询性能能够优于 FAST&FAIR 以及 wB+tree. 此外, 由于在底层结构上加入了 B+树结点的设计, HART 索引对于范围查询的支持预计能够大幅优于 WOART 索引.

(1) 读写吞吐

图 11 显示了各类索引在密集分布和聚簇分布下的平均读写吞吐.



(a) 密集分布 (b) 聚簇分布

图 11 HART 与其他索引在密集和聚簇分布下的吞吐对比

在密集分布数据集上, HART 的叶结点利用率最高, 因此在读写吞吐上均优于其他所有索引. 但在聚簇分

© 中国科学院软件研究所 http://www.jos.org.cn

布数据集上, HART 的读性能有明显的下降, 虽然高于 wB+树和 FAST&FAIR, 但低于 WOART. 这是因为 HART 底层是由 NVNode 构成的树形结构, 是对键分片的最后一字的划分, 在聚簇分布数据集上, HART 索引的高度为 1–3 层; 而在 WOART 中, 对应高度只有 1 层, 加上额外的叶子的访问, 因此 HART 的路径长度仍比 WOART 长一个结点. 为了进一步测试 HART 的点查询性能, 我们在聚簇分布数据集上改变点查询的次数并统计索引的查询延迟, 结果如图 12 所示. 可以看到, 在不同的点查询次数下, 在聚簇分布上 HART 索引的点查询性能都低于 WOART. 但 HART 相比于 WOART 的优势在于, 它可以很好地支持范围查询, 这得益于 HART 底层采用的有序结点结构. 在后面的范围查询实验中我们可以看到, WOART 的范围查询性能比 HART 要差 2–3 个数量级.

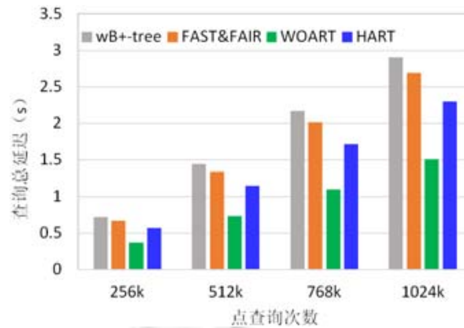


图 12 点查询时间延迟对比

(2) 范围查询性能

WOART 由于范围查询会进行回溯, 在 NVM 上更高的访问代价使得 WOART 的范围性能无法与其他索引相比, 执行时间与其他索引有 2–3 个数量级的差距(在我们的实验中, WOART 执行纯范围查询的时间为其他索引的 200–300 倍), 因此后面的范围查询结果中我们不再给出 WOART 的结果(差距太大).

图 13 给出了纯 NVM 上除 WOART 外各索引的范围查询性能对比. 实验继续使用聚簇分布数据集, 每次实验会查询总计 64 M 项数据. 从图 13 可以看出, FAST&FAIR 得益于结点的有序性, 范围查询性能最好; 而 wB+tree 虽然物理存储无序, 但其结点间的逻辑有序性也使其表现了较好的范围查询性能; HART 由于上层结点没有保留逻辑或物理有序, 进行范围查询时, 每个结点都需要把键读入 DRAM 中进行排序, 再进行输出, 因此范围查询性能低于 wB+tree 和 FAST&FAIR. 这是由 HART 所基于的 Radix 结构所决定的.

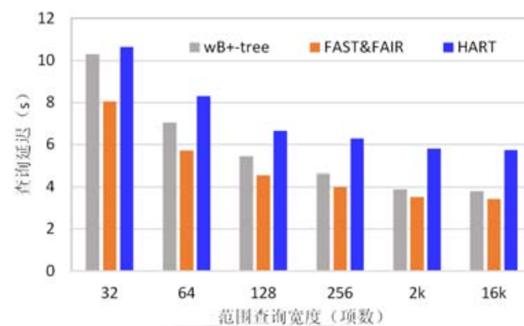


图 13 范围查询时间延迟对比

从能够满足范围查询性能需求的索引上来看, HART 的性能要高于 FAST&FAIR 与 wB+-tree. 一方面得益于上层 Radix 树结点的访问效率优势; 另一方面则是 NVNode 存储的是键分片, 键与头部能够在同一缓存行内刷回, 以相同的持久化代价能够刷回更多的数据, 即缓存行刷回次数更少, 因此写效率优于 wB+-tree. 但是由于子树间的维护的链表结构避免了结点回溯至 Radix 树层, 因此 HART 范围查询执行时间的趋势与 B+树类

型的索引相同, 并没有随查询宽度增加而增加.

尽管 HART 在范围查询上没有取得最好的性能, 但由于它在写性能和点查询性能上均优于 wB+tree 和 FAST&FAIR (如图 11 和图 12 所示), 因此总体上 HART 能够在写性能、点查询和范围查询之间取得更好的折中, 可以适应不同应用的存取需求, 不会出现像 WOART 这样无法很好地支持范围查询的问题.

(3) 不同 value 大小时的性能对比

在前面的实验中, 我们默认都使用了固定大小的 key 和 value. 接下来我们分析不同键值大小对于 HART 索引的性能影响. 由于已有的索引仅支持固定大小的 key, 因此在本实验中, 我们通过改变 value 的大小来验证索引在不同记录大小时的表现. 实验继续使用聚簇分布数据集, 并测试 3 组 value 大小: 8 B, 256 B 和 4 KB, 分布对应小记录、中记录和大记录的情况.

图 14 给出了不同 value 大小时, 4 个索引的查询性能对比, 此处的查询都是点查询. 我们发现, HART 的查询性能在中等记录大小(256 B)时最好, 而且优于 WOART. 这说明只要数据分布使得 HART 的上层索引有较好的路径压缩率, HART 索引的点查询性能可以取得更好的效果. 在所有情况下, HART 的查询性能均优于 wB+tree 和 FAST&FAIR, 表明 HART 的查询性能并不会因为记录大小的变化而出现急剧的衰退.

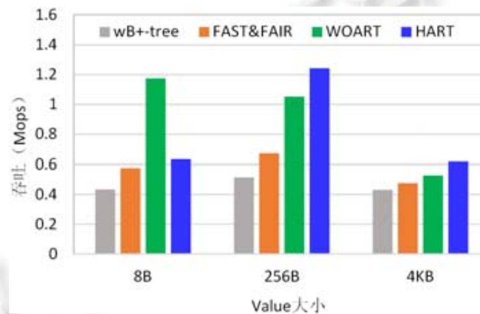


图 14 不同 value 大小时的查询吞吐对比

(4) 索引大小

表 2 给出了 4 个索引在密集分布和聚簇分布数据集上的大小对比. 实验中, 我们统计了写完 100 万条记录后的索引大小. 由于 Radix 树具有较严重的空间放大问题, 因此 HART 和 WOART 总体的空间代价都会高于 wB+tree 和 FAST&FAIR. 总体上, HART 和 WOART 可以看成是“空间换时间”的设计, 高查询性能的背后是空间存储代价. 在密集分布数据集上, 由于此时 HART 的叶结点空间利用率高, 因此 HART 索引的空间代价最低. 但是在聚簇分布数据集上, HART 索引的空间放大较严重, 达到了 10 倍以上. 分析其原因, 是因为目前 HART 索引的设计中未加入空间回收策略, 我们将在后续工作中对此加以改进.

表 2 索引大小对比 (MB)

索引	密集分布	聚簇分布
wB+-tree	28.69	28.70
FAST&FAIR	31.09	31.07
WOART	32.09	48.40
HART	14.88	277.37

另一方面, 由于 NVM 与 DRAM 相比具有高密度、大容量的优点, 因此 NVM 的空间代价在面向 NVM 的数据库系统中通常不会是个严重问题. 而且, 以空间换时间也是目前许多高效的索引结构采用的常见方法. 例如, RocksDB、HBase 等普遍使用的 LSM-tree 为了保证高写吞吐, 也同样具有 10 倍以上的空间放大率.

4 结束语

本文针对面向非易失内存的 Radix 树与 B+树类型索引各自的优势区间, 分析了适用场景, 并基于此提出了异构索引 HART. 为了大幅优化 Radix 树类型索引的范围查询, 我们提出了结合分配内存空间, 将零散的叶

© 中国科学院软件研究所 http://www.jos.org.cn

子存储在大小对 NVM 访问友好的数据结构中, 并通过兄弟指针维护一个持久化链表. 为了保证失败原子性, 对结点的分裂、路径的分裂、结点内存结构进行了设计, 实现了一个完整的索引结构.

我们对 HART 索引结构及其变种在仿真平台、Intel 傲腾持久性内存上分别进行了实验, 并与多个已有 NVM 索引进行了对比, 结果表明, HART 在纯 NVM 的内存架构下, 在对范围查询性能有需求的索引中, 有更优异的读写性能.

未来的研究工作将主要集中在 4 个方面: 首先, 从实验结果上来看, HART 在聚簇分布下的空间利用率较低, 空间放大较严重, 因此, 未来我们将进一步优化 HART 在聚簇分布下的内存分配与管理策略, 例如考虑使用延迟申请内存空间的策略, 或是暂存于兄弟结点的策略, 以提高结点内的空间利用率; 其次, 目前的 HART 版本未考虑并发控制技术, 因此在未来工作中, 我们将研究 HART 的并发控制策略, 例如使用多副本策略或者修改底层持久化链接的实现方式, 使 HART 能够适应大规模并发读写的应用场景; 第三, 目前的实验采用的是自定义的负载, 在未来工作中, 我们将考虑使用开放的测试基准如 YSCB 等进一步进行性能验证; 最后, 基于 DRAM+NVM 的混合内存也是目前学术界关注的架构, 在后续工作中, 我们将对 HART 进行改进, 使其能够适应异构混合内存架构.

References:

[1] Kultursay E, Kandemir M, Sivasubramaniam A, *et al*. Evaluating STT-RAM as an energy-efficient main memory alternative. In: Proc. of the ISPASS. 2013. 256−267.

[2] Mao M, Cao Y, Yu S, *et al*. Optimizing latency, energy, and relia-bility of 1T1R ReRAM through cross-layer techniques. IEEE Journal of Emerging and Selected Topics in Circuits and Systems, 2016, 6(3): 352−363.

[3] Yang J, Williams R. Memristive devices in computing system: Promises and challenges. ACM Journal on Emerging Technologies in Computing Systems, 2013, 9(2): Article No.11.

[4] Raoux S, Burr G, Breitwisch M, *et al*. Phase-change random access memory: A scalable technology. IBM Journal of Research and Development, 2008, 52(4-5): 465−480.

[5] Song S, Das A, Mutlu O, *et al*. Improving phase change memory performance with data content aware access, In: Proc. of the ISMM. 2020. 30−47.

[6] Izraelevitz J, Yang J, Zhang L, *et al*. Basic performance measurements of the Intel Optane DC persistent memory module. CoRR abs/1903.05714, 2019.

[7] Chen S, Jin Q. Persistent B+-trees in non-volatile main memory. Proc. of the VLDB Endowment, 2015, 8(7): 786−797.

[8] Kim W, Seo J, Kim J, *et al*. clfB-tree: Cacheline friendly persistent B-tree for NVRAM. ACM Trans. on Storage, 2018, 14(1): Article No.5.

[9] Chi P, Lee W, Xie Y. Adapting B+-tree for emerging nonvolatile memory-based main memory. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2016, 35(9): 1461−1474.

[10] Chen Q, Yeom H. Design of skiplist based key-value store on non-volatile memory. In: Proc. of the ICAC. 2018. 44−50.

[11] Lee S, Lim K, Song H, *et al*. WOART: Write optimal radix tree for persistent memory storage systems. In: Proc. of the FAST. 2017. 257−270.

[12] Ma S, Chen K, Chen S, *et al*. ROART: Range-query optimized persistent ART. In: Proc. of the FAST. 2021. 1−16.

[13] Ying Y, Huang K, Zheng S, *et al*. CANRT: A client-active NVM-based radix tree for fast remote access. In: Proc. of the ICA3PP. 2020. 433−447.

[14] Leis V, Kemper A, Neumann T. The adaptive radix tree: Artful indexing for main-memory databases. In: Proc. of the ICDE. 2013. 38−49.

[15] Liu R, Jin P, Wang X, *et al*. NVlevel: A high performance key-value store for non-volatile memory. In: Proc. of the HPCC/SmartCity/DSS. 2019. 1020−1027.

[16] Zhang X, Feng D, Hua Y, *et al*. A write-efficient and consistent hashing scheme for non-volatile memory. In: Proc. of the ICPP. 2018.

© 中国科学院软件研究所 http://www.jos.org.cn

[17] Zuo P, Hua Y, Wu J. Write-optimized and high-performance hashing index scheme for persistent memory. In: Proc. of the OSDI. 2018. 461–476.

[18] Wan H, Li F, Zhou Z, *et al*. NVLH: Crash-consistent linear hashing for non-volatile memory. In: Proc. of the NVMSA. 2018. 117–118.

[19] Xia F, Jiang D, Xiong J, *et al*. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In: Proc. of the ATC. 2017. 349–362.

[20] Venkataraman S, Tolia N, Ranganathan P, *et al*. Consistent and durable data structures for non-volatile byte-addressable memory. In: Proc. of the FAST. 2011. 61–75.

[21] Hwang D, Kim W, Won Y, *et al*. Endurable transient inconsistency in byte-addressable persistent B+-tree. In: Proc. of the FAST. 2018. 187–200.

[22] Yang J, Wei Q, Wang C, *et al*. NV-tree: A consistent and workload-adaptive tree structure for non-volatile memory. IEEE Trans. on Computers, 2016, 65(7): 2169–2183.

[23] Oukid I, Lasperas J, Nica A, *et al*. FPtree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In: Proc. of the SIGMOD. 2016. 371–386.

[24] Jeong J, Hong J, Maeng S, *et al*. Unbounded hardware transactional memory for a hybrid DRAM/NVM memory system. In: Proc. of the MICRO. 2020. 525–538.

[25] Zhou X, Shou L, Chen K, *et al*. DPtree: Differential indexing for persistent memory. Proc. of the VLDB Endowment, 2019, 13(4): 421–434.

[26] O'Neil P, Cheng E, Gawlick D, *et al*. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351–385.

[27] Han S, Jiang D, Xiong J. LightKV: A cross media key value store with persistent memory to cut long tail latency. In: Proc. of the MSST. 2020.

[28] Liu J, Chen S, Wang L. LB+-trees: Optimizing persistent index performance on 3DXPoint memory. Proc. of the VLDB Endowment, 2020, 13(7): 1078–1090.

[29] van Renen A, Vogel L, Leis V, *et al*. Building blocks for persistent memory. The VLDB Journal, 2020, 29(6): 1223–1241.

[30] Hirofuchi T, Takano R. A prompt report on the performance of Intel Optane DC persistent memory module. IEICE Trans. on Information & Systems, 2020, 103-D(5): 1168–1172.

[31] Schwalb D, Berning T, Faust M, *et al*. nvm_malloc: Memory allocation for NVRAM. In: Proc. of the ADMS@VLDB. 2015. 61–72.

刘睿诚(1995－), 男, 博士生, 主要研究领域为面向非易失性内存的存储技术.

罗永平(1997－), 男, 博士生, 主要研究领域为面向非易失性内存的存储技术.

张俊晨(1996－), 男, 硕士, 主要研究领域为面向非易失性内存的内存索引.

金培权(1975－), 男, 博士, 副教授, 博士生导师, CCF高级会员, 主要研究领域为数据库系统及应用, 大数据存储与管理.

© 中国科学院软件研究所 http://www.jos.org.cn