软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(5):2321−2341 [doi: 10.13328/j.cnki.jos.007198] [CSTR: 32375.14.jos.007198]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

PLTree: 一个高性能持久化内存学习索引*

张志国[1,2], 谢钟乐[1,2], 陈 珂[1,2], 寿黎但[1,2]

[1](区块链与数据安全全国重点实验室 (浙江大学), 浙江 杭州 310027)

[2](浙江大学 计算机科学与技术学院, 浙江 杭州 310027)

通信作者: 谢钟乐, E-mail: xiezl@zju.edu.cn

摘 要: 持久化内存 (persistent memory, PM) 作为主存的补充和替代, 为数据存储提供了相对较低的价格成本, 并且保证了数据的持久化. 为 PM 设计的传统结构索引 (如 B+树等) 未能充分利用数据分布特点来发挥索引在 PM 上的读写性能. 最近的研究尝试利用学习索引的数据分布感知能力提升索引在 PM 上的读写性能并实现持久化. 但在面对真实世界的数据时, 现有基于 PM 的持久化学习索引的数据结构设计会导致额外的内存访问, 从而影响读写性能. 针对 PM 学习索引在面对真实数据时读写性能下降的问题, 提出一种 DRAM/PM 混合架构的学习索引 PLTree. 它通过以下方法提升在 PM 上的读写性能并减轻数据分布颠簸对性能的影响: (1) 使用两阶段方法构建索引消除内部节点的局部搜索, 减少 PM 的访问. (2) 利用模型搜索来优化 PM 上的查找性能并通过在 DRAM 存储元数据加速查找. (3) 根据 PM 的特性设计了日志式分层溢出缓存结构, 优化写入性能. 实验结果表明, 在不同数据集上, 与现有的持久化内存索引 (APEX, FPTree, uTree, NBTree 和 DPTree) 相比, PLTree 在索引构建性能上平均提升了约 1.9–34 倍; 单线程查询/插入性能平均提升了约 1.26–4.45 倍和 2.63–6.83 倍; 在多线程场景, 查询/插入性能最高提升了约 10.2 倍和 23.7 倍.

关键词: 学习型索引; 持久化内存; 持久化内存索引; 数据库

中图法分类号: TP311

中文引用格式: 张志国, 谢钟乐, 陈珂, 寿黎但. PLTree: 一个高性能持久化内存学习索引. 软件学报, 2025, 36(5): 2321–2341. http://www.jos.org.cn/1000-9825/7198.htm

英文引用格式: Zhang ZG, Xie ZL, Chen K, Shou LD. PLTree: High-performance Learning Index for Persistent Memory. Ruan Jian Xue Bao/Journal of Software, 2025, 36(5): 2321–2341 (in Chinese). http://www.jos.org.cn/1000-9825/7198.htm

PLTree: High-performance Learning Index for Persistent Memory

ZHANG Zhi-Guo[1,2], XIE Zhong-Le[1,2], CHEN Ke[1,2], SHOU Li-Dan[1,2]

[1](State Key Laboratory of Blockchain and Data Security (Zhejiang University), Hangzhou 310027, China)

[2](College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

Abstract: Persistent memory (PM), serving as a supplement and potential replacement for main memory, offers a lower cost for data storage while ensuring data persistence. However, traditional index structures tailored for PM like B+ trees fail to fully exploit the distribution characteristics of data for optimizing reading and writing performance on PM. Recent research endeavors have sought to enhance indexes' reading and writing performance on PM and support index persistence through the data distribution awareness of learning indexes. Nonetheless, existing designs of persistent learning index structures suffer from additional PM accesses and poor performance when confronted with real-world data. To address the performance degradation of persistent learning indexes in the face of real data distributions, this study proposes a learning index PLTree, a DRAM/PM hybrid architecture. PLTree optimizes reading and writing performance under real data distributions through the following approaches: (1) a two-stage approach to construct the index, eliminating last-mile search in internal nodes and reducing the access of PM, (2) model-based search for efficient query performance on PM and

* 基金项目: 浙江省尖兵研发攻关计划 (2024C01021); 浙江省科技创新领军人才计划 (2023R5214)
收稿时间: 2023-12-11; 修改时间: 2024-01-21, 2024-03-10; 采用时间: 2024-03-28; jos 在线出版时间: 2024-06-14
CNKI 网络首发时间: 2024-06-17

accelerated query by leveraging metadata in DRAM, and (3) a log-based hierarchical overflow buffer structure tailored to PM characteristics to optimize writing performance. The results show that, compared with the existing persistent memory indexes (APEX, FPTree, uTree, NBTree, and DPTree), PLTree achieves significantly better performance in index construction 1.9× to 34× across various datasets. In single-threaded scenarios, PLTree exhibits an average query and insertion performance improvement of 1.26× to 4.45× and 2.63× to 6.83×, respectively. In multi-threaded scenarios, PLTree surpasses the baseline by up to 10.2× and 23.7× in query and insertion performance, respectively.

Key words: learning index; persistent memory (PM); persistent memory index; database

随着数据规模的快速增长, 现代数据库受到 DRAM 高成本和低容量的限制[1]. 持久化内存 (persistent memory, PM)[2]的出现为解决这一问题提供了新的解决思路. 同时, PM 自身的特性也给基于 PM 的数据库研发带来了新的挑战. 例如, 在目前唯一商用的 PM 产品 Optane DCPMM 上进行应用开发时需要考虑以下因素[3]: 不对称的读写延时 (写入较慢); 有限的读写带宽资源; CPU 缓存行的访问粒度 (64 B) 与 3D-XPoint 的存储介质访问粒度 (256 B) 不匹配导致的读写放大; 持久化操作开销等. 近年来, 研究者针对 PM 特性优化了数据库索引结构, 涵盖了基于树结构[4−14]和哈希结构[15−20]等多种类型的持久化内存索引技术. 然而, 这些基于传统数据结构的研究在设计持久化内存索引时未充分考虑数据的分布模式, 导致索引的性能不佳. 例如, 在基于 B+树结构的持久化内存索引中, 为了匹配 PM 介质的访问粒度, 叶节点通常设置为固定大小 (如 256 B). 索引高度与数据规模及叶节点大小有关, 由于未充分利用数据分布特点, 随着数据量的增加, 叶节点数量也随之增加, 这导致需要构建更高的树来完成数据索引, 导致性能下降.

最近提出的学习索引技术[21]能够根据数据分布特征构建更为扁平的索引结构, 从而显著降低搜索成本. 其性能优势主要体现在两方面[22]: 一方面, 学习索引相比传统 B+树能创建更大的数据节点, 容纳更多数据, 进而减少索引层级和查询路径长度; 另一方面, 学习索引通过内部模型预测键值大概的存储位置, 在节点内查找时只需在预测位置附近查找, 相比于遍历整个节点产生的开销, 利用简单模型定位的计算开销要小得多. 因此, 在 PM 上构建基于数据分布的学习索引成为进一步提升基于 PM 的数据库索引性能的研究方向之一. 然而, 当前主流学习索引多基于 DRAM 设计, 直接应用于 PM 时会产生额外开销[23]. 最近的工作[1,24,25]针对这一问题进行了探索, 对学习索引结构进行了调整以适应 PM 特性, 实现学习索引的持久化, 依据数据分布特点建立更扁平的索引结构, 借助模型搜索来减少不必要的 PM 访问. 然而, 这些尝试尚未完全解决移植后索引存在的性能问题:

首先, 学习索引的性能与数据分布特征密切相关, 在真实负载上读写性能较差. 有研究[26]指出, 当从合成数据集切换到真实数据集时, 学习索引的性能显著下降. 因为真实数据集通常具有非平滑和非线性的特性, 可能导致叶节点内数据冲突增加, 即不同键值可能定位到同一位置. 为解决此问题, APEX[1]和 PLIN[24]在叶节点无法容纳冲突时需要采用溢出缓存结构存储冲突数据, 但这会带来额外的访问开销, 例如查找时不论目标键是否存在于叶节点, 都必须先在叶节点访问至少 1 个 PM 块 (256 B), 然后决定是否继续搜索溢出缓存. 当目标键不存在时, 需要完整遍历叶节点预测位置和对应的溢出缓存区域, 导致读性能的下降. 同样, 在插入操作中, 也需要在叶节点和溢出缓存中进行键的唯一性检查, 最坏情况下需完整遍历预测位置和缓存区域, 影响写入性能.

其次, 由于模型预测误差引起的局部搜索也会引起额外的 PM 访问. 学习索引中的局部搜索是指在查找时, 由于模型预测不一定能完美给出准确的预测位置, 需要在预测位置的附近再进行一次搜索. 以 PLIN 为例, 其构建内部节点时使用的分段线性逼近算法 (OptimalPLR)[27]基于固定误差对索引各层数据进行分段, 索引内部节点中的模型存在预测误差, 需要在误差范围内进行局部查找来定位下一层的子节点, 这一操作增加了内存的访问. 而 APEX 和 APLI[25]采用自顶向下的方法构建索引, 内部节点中没有局部搜索问题, 但是无法保证叶节点中模型的误差上限. 另外, 在面对难以用简单模型拟合的局部键空间时, 可能会导致更长的查找路径.

由于 PM 的访问开销比 DRAM 大, 在 PM 中, 上述问题对性能的影响会被放大, 并最终影响持久化学习索引的读写性能. 为此, 本文提出了持久化学习索引 PLTree. 它针对 PM 的特性进行设计, 确保崩溃一致性、支持索引恢复, 优化了面对真实数据时的读写性能. 本文的主要贡献如下: (1) 优化局部搜索, 减少 PM 访问次数. PLTree 采用两阶段索引构建方法, 在索引构建过程中消除了由于内部节点中模型预测误差导致的局部搜索, 减少了 PM 访

问. 叶节点中的数据被划分为地址按 256 B 对齐的块, 将叶节点内的局部查找限制在一个 PM 块内完成. (2) 实现了索引的并发操作, 并使用 DRAM 存储元数据来维护叶节点和溢出缓存结构中键的指纹信息, 加速查询操作, 避免了频繁的 PM 访问, 节约有限的 PM 带宽. (3) 设计了写优化的分层缓存结构, 以解决真实数据负载下索引写入性能下降的问题. 在叶节点中为每个块动态设置了一个分层溢出缓存. 当块满时, 将其中的数据批量写入溢出缓存, 减小写放大. 此外, 针对索引的插入操作进行了优化. PLTree 中支持存储键重复的数据, 插入数据时键唯一性检查仅在叶节点层面进行. 一旦完成叶节点的查询, 就不再查询溢出缓存, 避免了额外的 PM 访问, 从而提高索引写入性能. (4) 在真实的 PM 设备上实现了 PLTree, 并进行了全面的对比和分析. 将 PLTree 与其他先进的持久化内存索引进行了比较, 以验证索引设计的先进性.

1 相关工作

1.1 持久化内存索引

为了提升索引在 PM 上的性能, 之前的工作从多个角度提出了优化方案: (1) 不同的数据存储方式. 一些索引 (如 BzTree[14], Dash[18]等) 选择将索引完全存储在 PM 中, 便于索引的恢复, 但在索引操作时 (如元数据的更新) 可能会产生额外 PM 的读写开销. 而另一些索引 (如 FPTree[6], NBTree[8], uTree[13]等) 采用了 DRAM/PM 混合存储模式, 只将关键数据持久化, 将频繁访问的索引结构放于 DRAM 中, 从而减少了 PM 的访问开销. 然而, 这种方式需要在恢复时重构索引, 牺牲了一部分恢复性能, 并需要设计复杂的恢复策略以实现快速恢复. (2) 数据结构的优化. 文献 [5−7,14] 采用了无序的叶节点存储数据, 叶节点中数据以日志形式追加写入, 无需保证叶节点内记录的有序性, 避免了数据移动的开销, 但查找时需要在叶节点内线性遍历, 影响搜索性能. 文献 [9,11] 通过记录键在无序叶节点内的排序顺序改善搜索性能. 文献 [6−8,11,12,18] 在索引中存储了键的指纹, 使用 1 字节指纹作为键的摘要信息加速搜索, 避免不必要的数据探测操作. 文献 [10] 采用了叶节点压缩方法, 减少了根到叶的路径长度和遍历开销. 文献 [17] 根据 Optane DC 持久化内存硬件特性, 将叶节点大小的设置为 256 B 的倍数以提高带宽利用率. 文献 [15,17,18] 通过限制探测距离, 确保查找性能. 文献 [12,20] 设计了分层的数据结构, 数据先存入 DRAM 缓存结构, 之后再批量写入 PM, 优化写入性能. (3) 其他. 文献 [7,10,18] 使用 SIMD 指令增加并行处理能力, 加速读操作. 文献 [10−13,19] 通过对内存分配进行优化降低了持久化开销.

近期研究[1,24,25]尝试运用学习索引替代传统索引结构, 利用数据的分布特点构建基于持久化内存的学习索引. 其中, APEX 和 APLI 采用 DRAM/PM 混合架构. APEX 是 ALEX[28]的持久化版本, 针对 PM 进行了改进. 它将叶节点视为哈希表, 将叶节点模型视为哈希函数, 通过探测和存储 (probe-and-stash) 解决叶节点中多个键预测位置相同时引发的冲突. 在叶节点中查找时, APEX 将数据的探测距离限制在 16 个槽位 (slot) 内, 若未找到目标键则继续搜索溢出桶. 为减少数据移动产生的持久化开销, APEX 采用间隔数组 (gapped array) 存储插入数据, 并将频繁访问的元数据存储在 DRAM 中, 以减少额外 PM 访问. APLI 则采用链表将数据节点 (256 B) 存储在 PM 中, 并使用存储在 DRAM 中的 ALEX 索引结构作为上层节点来索引数据节点中的最小键. 同时, 它还在 DRAM 中设置了一个哈希表用于存储热点数据, 以加速热点数据的访问. PLIN 是一个纯 PM 架构的学习索引, 它采用了 PGM[29]的索引构建方法, 根据 PM 介质访问粒度将数据节点划分为多个 256 B 的块, 并为每个块动态设置了树结构的溢出缓存 (Fast&Fair[4]), 在块中预留空间, 以块内无序的方式存储新加入的数据, 避免数据的移动开销. PLIN 将叶节点中的查找操作限制在一个 PM 块内, 以减少额外的 PM 访问.

1.2 学习索引

学习索引根据数据分布构建, 其构建过程实际上是通过机器学习模型来近似数据-位置的累积分布函数 (cumulative distribution function, CDF). 学习索引模型中存储了键到位置的映射关系, 理想的情况下通过模型推断能够使索引查找复杂度到达 O(1). 然而, 单个模型通常难以很好地拟合复杂的数据分布, 因此学习索引一般需要训练多个机器学习模型, 组成与 B+树类似的树形结构来实现 CDF 的拟合: 上层模型接收键作为输入用于选择下一层模型, 逐层缩小查找范围直到找到叶节点, 接着通过叶节点模型来预测输入键在叶节点中的估计位置. 为了保

证计算和学习的效率, 大多数学习索引通常采用多个简单模型的组合来近似 CDF.

学习索引中存在局部搜索的问题, 即模型只能预测键的大概位置, 需要在预测位置附近进行局部搜索以确定准确位置. 在一些学习索引中 (如 APEX, ALEX, FITing-Tree[30], ALERT[31]等) 模型误差导致的局部搜索只存在于叶节点, 而其他一些索引 (如 PLIN, PGM, XIndex[32], FINEdex[33]等) 局部搜索不仅存在于叶节点还存在于内部节点. 为改善局部搜索性能, PGM、FITing-Tree 以及 ALERT 等工作通过设定模型误差上限, 并结合二分查找策略来优化. ALEX 则采用指数搜索方式提升叶节点内的局部查找效率. APEX 和 PLIN 限制了叶节点内的查找距离, 从而避免过多的 PM 访问. 此外, LIPP[34]和 DILI[35]提出了消除内部节点与叶节点中局部搜索的方法. LIPP 采用了最小化冲突的模型消除局部搜索, 当同一位置产生多个键冲突时, 创建一个新的子节点来覆盖冲突的数据, 将相同位置的键存入子节点中, 并将原先的键值对设置为指向子节点的指针, 以较长的遍历路径为代价消除局部搜索. DILI 通过两阶段索引构建和局部优化的方法消除了内部节点和叶节点内的局部搜索.

目前, 构建学习索引主要有两种方法: (1) 自底向上法. 该方法设定一个固定的误差, 使用分割算法将有序排列的键空间切割为若干个互不相交的有序子集, 作为叶节点中的数据分区. 在构建上层的内部节点时, 将所有叶节点分段边界键汇总并作为新的划分对象, 递归使用分割算法将边界键划分到上层的不同内部节点中, 直到形成根节点[24,29]. 此外, 还可以借助叶节点边界键构建 B+树或 Radix 树来索引叶节点[30,31]. (2) 自顶向下法. 该方法从根节点开始, 学习整个数据集 CDF 的大致轮廓, 然后判断是否进一步向下拆分子节点. 若节点需要向下拆分, 则使用多个子节点模型来细化分割 CDF 曲线, 拟合其局部数据分布, 并在该节点内训练一个键到子节点的模型. 递归执行直至满足叶节点的划分条件为止, 然后训练一个模型来预测叶节点中键的位置.

例如, PGM、PLIN 等采用的是自底向上的索引构建方法. 该方法的优点在于: (1) 可以通过一次性顺序扫描数据集快速完成数据分区与模型拟合. (2) 依据预设的误差上限来划分每个节点, 确保在最差情况下, 节点内模型的误差仍能得到控制. (3) 能够根据完整的数据集划分 CDF, 从而更好地理解关键字的分布, 并据此合理地划分子节点. 该方法的缺点为: 节点内的模型存在预测误差, 因此查找操作需要在内部节点中进行局部搜索才能确定下一层节点, 增加了查找开销. 相比之下, 自顶向下方法在构建索引时, 依据预设分支数量精确划分内部节点, 因此查找子节点时, 无需在内部节点中执行局部搜索. 但是构建索引时通常需要预先设定索引的分支数量, 不够灵活且得到的数据分区可能会最终降低叶节点模型的准确性. 以 ALEX 和 APEX 为例, 它们采用自顶向下的方法构建索引, 通过成本模型估计索引的查找/插入平均延时, 并选择合适的子节点分割数量, 利用均匀数据分区的方式向下分割节点. 自顶向下的分割方法无法在学习阶段感知数据分布的整体情况. 例如, ALEX 和 APEX 在节点向下拆分的过程中, 将子节点的数量固定为 2 的指数, 这意味着在构建索引时, 不能根据实际数据分布情况动态调整最优的分支数量分割 CDF. 这可能会导致索引布局不能很好地近似某些特定的数据分布. 另外, 自顶向下的方法也无法确保叶节点内预测误差的上限. 如果分区粒度过粗, 叶节点中的线性模型可能无法很好地拟合某些范围内的 CDF. 在这种情况下, ALEX 的叶节点中可能产生更大的误差, 导致更大范围的局部搜索; 而 APEX 的叶节点中可能会引发更多的冲突, 导致更多数据被写入溢出桶中.

为解决局部搜索的问题, DILI 采用了两阶段构建索引的策略: 首先, 在第 1 阶段, 根据数据分布使用贪心算法自底向上计算内部节点和叶节点的搜索成本, 并在所有可能的索引布局中寻找一个既能使搜索成本最小又能有效体现键分布特点的最优索引布局方案. 然后, 在第 2 阶段, 根据得到的索引布局, 采用自顶向下的方法构建索引. 根据布局中每个节点的分支数量均匀地划分子节点范围, 通过自顶向下方法构建索引, 消除内部节点的局部搜索开销. 然而, DILI 在构建索引布局时需要遍历所有可能布局以确定搜索成本最低的那一种, 这导致了较高的计算开销. 同时, 每层使用的贪婪合并算法代价较高, 其时间复杂度为 $O(n\log_2 n)$. 此外, DILI 在构建过程中, 从底层开始创建线性回归模型, 并使用线性回归计算模型参数. 每次合并节点时, 都需要计算节点中所有数据 (键-位置) 之间平方差的最小值, 以调整模型的斜率和截距. 这导致 DILI 的索引构建时间较长.

2 索引结构

PLTree 是 DRAM/PM 混合架构的持久化学习索引, 其结构如图 1 所示, 其中内部节点、叶节点以及溢出缓存

均存储在 PM 中, 元数据则存储在 DRAM 中负责维护叶节点和溢出缓存中键的指纹信息, 加速索引的查找. 内部节点和叶节点存储了线性模型的信息. 在内部节点中 PLTree 消除了局部搜索, 因此可以通过模型定位指向下一层子节点的槽位的准确位置. 叶节点中, 模型以数据块为粒度预测键值对存储的位置.

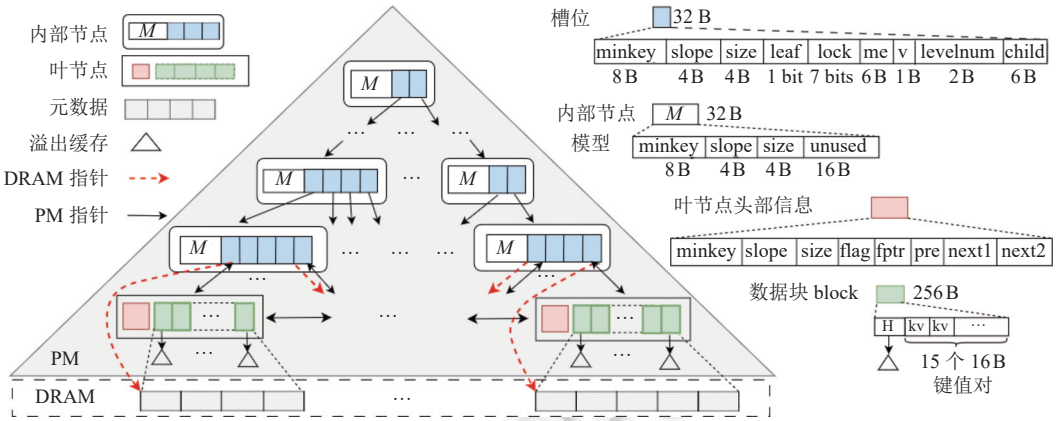


图 1 索引结构

2.1 内部节点

如图 1 所示, 内部节点存储在 PM 中. 其结构主要由两部分构成: 头部模型 *M* 和槽位数组. 内部节点模型根据键预测子节点对应槽位的位置, 头部模型包括以下内容: 最小键 minkey (8 B)、单精度浮点数 slope (4 B) 表示模型的斜率, 以及 size (4 B) 表示槽位数组的大小, 也代表节点的分支数量.

为了减少 PM 访问, 每个槽位都存储了其指向的子节点的模型信息. 因此, 在搜索过程中, 除了访问根节点需要获取节点头部的模型信息外, 在向下搜索时无需访问子节点的头部信息, 通过模型准确定位, 每次只需访问内部节点的一个槽位. 内部节点结构设计考虑了缓存友好性, 将头部模型和每个槽位大小统一设置为 32 B, 并确保内部节点数据结构按 256 B 对齐. 根节点的大小最大限制为 256 B, 因此每个内部节点的访问只会读取 1 个 PM 块 (256 B), 且每次访问的数据范围不会超过一个缓存行 (64 B). 在当前的 AMD64 和 Intel® 64 架构中, 指针的有效位为 48 位, 因此在槽位中只使用 6 B 存储指针信息. 其中 child (6 B) 为子节点的指针. 每个槽位中的 leaf (1 bit) 表示所指向的节点是否为叶节点, 若 leaf 为 0, 则 child 指向内部节点, 并且槽位中的 minkey、slope 和 size 字段为所指向的内部节点线性模型的参数. 当 leaf 为 1 时, child 指向叶节点, 此时 minkey、slope 和 size 字段是所指向的叶节点中线性模型参数. 此外, 字段 me、v、levelnum 和 lock 只有在子节点为叶节点时有效, me (6 B) 为指向 DRAM 中元数据数组的指针, v (1 B) 存储所指向叶节点的版本信息, 用于系统崩溃后的索引恢复, levelnum (2 B) 存储了所指向叶节点中所有溢出缓存的总层数, lock (7 bits) 为节点扩展时的叶节点锁.

2.2 叶节点

叶节点的结构如图 1 所示, 叶节点之间通过指针组成双向链表, 以支持范围查找和索引的恢复. 叶节点由头部信息和数据块 (block) 数组两部分组成: (1) 头部信息存储了叶节点的模型参数信息, 包括斜率 slope、叶节点最小键 minkey 和数据块数组的长度 size. 此外, 头部信息还包括一个指向前一兄弟叶节点的指针 pre; 两个备选兄弟节点指针 (next1, next2) 用于指向下一个叶节点; 字段 flag 用于切换两个备选兄弟节点指针, 实现无日志叶节点修改; 字段 fptr 为指向父节点槽位的指针, 用于恢复时的一致性检查. (2) 数据块数组地址按 256 B 对齐. 其中各个 block 之间的键保持有序, 而 block 内部的键无需有序排列. 每个 block 的大小为 256 B, 其中头部 H (16 B) 中存储了指向溢出缓存的指针, 当 block 不存在溢出数据时, 此指针为空. 剩余空间可以存储最多 15 个 16 B 的键值对, 键值对由 8 B 的键和 8 B 的数据组成. 叶节点中数据的加载采用了 NVM 感知的数据放置策略[24], 即采用以 block 为粒度的线性模型预测 block 的位置, 并将键值对以无序方式写入其中. 叶节点模型预测粒度设置为 256 B 的好处是可以

将叶节点中局部搜索范围限制在 1 个 PM 块内.

2.3 溢出缓存

在面对分布不均匀的真实负载时, 学习索引往往无法很好地适应数据的分布, 这会导致叶节点中出现大量冲突, 进而降低读写性能. 特别是在 PM 中, 性能下降更为明显. 使用 face 数据集[26]进行测试时, 发现在 APEX 和 PLIN 中初始化加载 1 亿个数据后, 约 30%–50% 的数据会被写入溢出缓存, 从而导致大部分操作在访问叶节点之后, 需要额外访问溢出缓存. 因此, 在面对用线性模型难以拟合的真实负载时, 溢出缓存将成为持久化学习索引的性能瓶颈. 为了解决面对真实数据负载时持久化学习索引拟合数据分布不佳导致的性能问题, 本文受到先前的工作[12,20,36]启发, 采用了一种类似 LSM 树 (log-structured merge tree)[36]的分层日志式的写优化结构来存储由于冲突而溢出的数据, 优化大量数据写入溢出缓存时的性能.

溢出缓存只有在产生数据溢出的情况下才会创建, 并根据溢出的数据量动态创建新的缓存数据层. 如图 2 右所示, 缓存结构中的每一层都包含一个地址按 256 B 对齐的条目 (entry) 数组, 每向下一层, 该数组的长度扩大一倍. 每个 entry 可以存储 48 个键值对, 其中存储的键无序且允许存在重复. 在缓存结构每一层中, 为每个 entry 设置了一个 8 B 的 size 字段, 用于记录该 entry 中已经写入键值对的有效数量. 一旦某个 entry 完成批量写入操作, 其对应的 size 会被更新, 上一层对应 entry 的 size 则被置为 0. 其中 size 字段通过 8 B 原子写操作写入到 PM 中, 以确保崩溃一致性.

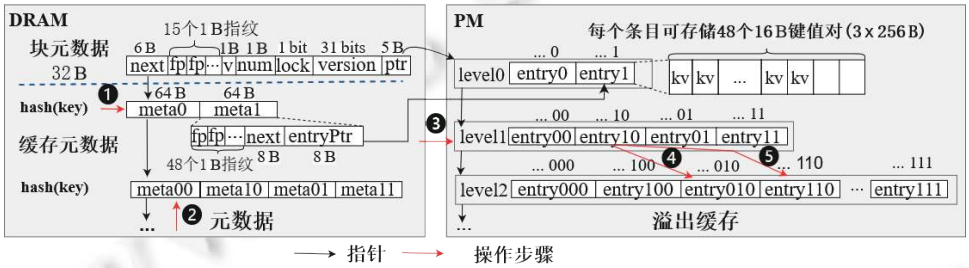


图 2 元数据结构 (左) 和溢出缓存结构 (右)

不同于 LSM, 缓存结构在每个层级的 entry 数组中采用哈希表进行索引. 哈希值冲突的键会被放置于同一个 entry 中. 由于 entry 数组的长度设置为 2 的幂次方, 因此在第 n 层, 需要使用键哈希值的低 n+1 比特来确定 entry 在数组中的位置. 在第 n 层中查找键时, 需对哈希值的低 n+1 比特位进行左右翻转以确定实际存储的 entry. 因为条目数组中的 entry 按照键哈希值低 n+1 比特左右翻转后的顺序排列. 如图 2 右所示, 在第 2 层需要使用键哈希值的 0–2 比特位进行索引, 二进制数 000 经过左右翻转后为 000, 代表键在 entry 数组中位置为 0; 二进制数 100 经过翻转后为 001, 代表键在 entry 数组中位置为 1. entry 数组中这种比特位翻转排序方式确保了上一层同一 entry 中的数据能够被尽量写入下一层地址连续的两个 entry 之中. 举例来说, 如图 2 右所示, 当第 1 层的 entry10 满时, 其中的数据会被写入第 2 层的 entry010 和 entry110 中. 这样可以使数据能够批量写入连续的内存空间, 充分利用持久化内存顺序写具有较高性能的特点, 提高写入带宽的利用利率, 优化写入性能.

此外, 在将数据写入下一层 (n+1 层) entry 时, 需要先计算第 n+1 层的 entry 中是否有足够的空闲位置. 若有充足空间, 第 n 层的数据会以日志追加的方式从左到右写入 n+1 层的 entry 中, 确保新数据始终位于 entry 中右侧. 当 n+1 层 entry 空间不足时, 需要将第 n+1 层 entry 中的数据批量写入 n+2 层, 若下层空间仍然不够, 则继续按照上述方法递归执行. 在这一过程中, 上层 entry 中重复的键值对将会被清理, 只保留最新的键值对, 再将其批量写入下一层. 实验研究表明[37], 对于超过 256 B 的访问, 使用非临时写入指令具有更低的延时. 因此, 为了提高批量写入的性能, 每次向下层写入数据时, 均采用非临时写入指令进行批量写入.

2.4 元数据

PLTree 在 DRAM 中存储了叶节点中每个 block 以及对应的溢出缓存中键的指纹信息和每个 block 的锁信息. 如图 2 左所示, 元数据分为块元数据和缓存元数据, 具体结构如下所述.

块元数据存储在 DRAM 中, 以 64 B 地址对齐的方式组织成数组. 每个块元数据的大小为 32 B, 其中包含以下字段: (1) 字段 next (6 B) 存储了指向缓存元数据的地址. (2) 字段 lock (1 bit) 和字段 version (31 bits) 用于实现索引的乐观读写并发控制. (3) 字段 v (1 B) 存储了当前元数据的版本用于索引恢复时元数据的重建. (4) 字段 num (1 B) 用于记录叶节点中对应 block 中存储的键值对数量. 当 block 中插入一个新的键值对时, num 原子加 1; 当 block 中的数据被清空时, num 原子置为 0. (5) 字段 ptr (5 B) 用于存储指向 PM 中缓存结构的地址 (由于缓存结构地址按 256 B 对齐, 地址的低 8 位为 0, 因此 ptr 只需使用 5 B 存储地址). (6) 剩余的空间 (15 B) 用于存储块数据中键的指纹信息.

与溢出缓存一样, 缓存元数据只有在产生数据溢出时才会被创建, 根据溢出数据的多少动态创建元数据层. 每一层的缓存元数据以 64 B 地址对齐的方式组织成 meta 数组, 每个 meta 的大小为 64 B. 其中包含以下字段: entryPtr (8 B) 存储了溢出缓存中的对应 entry 的持久化内存地址, next (8 B) 存储了指向下一层的元数据的指针, 剩余的 48 B 用于存储溢出缓存中对应 entry 中键的指纹信息.

3 索引操作

3.1 索引的构建

为了消除内部节点中的局部搜索, 受到 DLIL 和 PGM 的启发, PLTree 采用结合自底向上和自顶向下方法构建索引. 首先, 通过自底向上的方法构建索引布局, 根据完整的数据集划分累积分布函数, 能够更好地根据数据分布划分内部节点, 内部节点展开分支数量按照键的局部分布特点合理设置. 其次, 采用算法复杂度为 O(n) 的分段算法 FSW[38]进行键空间的分段划分, 通过设定合理的分段误差, 可以在保证索引布局合理性的同时, 加快索引的构建速度. 最后, 结合了自顶向下的方法构建索引, 消除了内部节点中由局部搜索导致的额外 PM 访问开销.

具体的做法是: (1) 采用自底向上的方法根据数据分布获得索引的布局 (每一层分段的键集合和每个分段范围包含的子节点数), 在索引每一层使用 FSW 算法对关键字升序排列的键空间进行分段. FSW 算法是一个时间序列分段算法, 具有线性的时间复杂度和常量的空间开销, 它使用贪心思想在满足误差的前提下划分键空间, 使每个分段内的数据点可以被近似为一条直线, 并且尽可能包含更多的数据点. (2) 得到索引布局后使用自顶向下的方法进行索引构建, 在内部节点中根据索引布局得到的分支数将节点范围平均划分为若干个槽位, 每个槽位的两个分段端点代表了其指向的子节点的数据范围, 然后采用线性插值的方法计算每个子节点线性模型参数: 通过槽位的两个分段端点 (minkey 和 maxkey) 与分支数 n 来计算下层子节点的线性模型的斜率 slope=n/(maxkey – minkey). 在下层子节点中, 每个槽位的范围为 (minkey+k/slope, minkey+(k+1)/slope], 其中 k 为槽位的序号且 $n > k \geqslant 0$. 因此在内部节点中查找键 key 时可以通过线性模型 pos=⌊(key – minkey)·slope⌋ 预测节点中槽位的准确位置来定位下层子节点, 而无需在节点中进行横向的局部搜索, 避免了额外的 PM 访问. 其中该线性模型的斜率为 slope, 截距为 – minkey·slope.

图 3 是索引操作步骤, 为方便展示, 叶节点每个数据块大小设置为 3, $M(a, b)$ 代表第 a 层的第 b 个节点的模型.

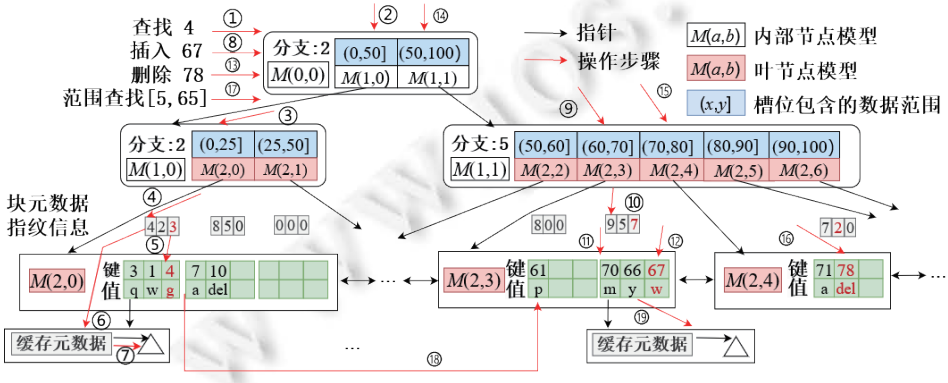


图 3 索引操作步骤

3.2 点查找

如图 3 红色箭头所示, 点查找操作先在索引内部节点中查找给定键, 确定键所在的叶节点地址和元数据地址 (①②③). 当查找到达叶节点的上一层内部节点时, 根据该内部节点存储的叶节点的模型参数计算出目标键在叶节点 block 数组中的位置和对应的块元数据地址. 由于避免了局部查找, 每层内部节点的查找只需要访问 32 B 的数据, 提升了缓存命中率, 每个内部节点中只需访问一个 PM 块, 节省了 PM 带宽资源. 在叶节点查找键的过程中, 首先读取块元数据中的指纹 (④). 如果存在匹配的指纹, 则继续在叶节点 block 中比较该位置处的键 (⑤). 如果元数据的指纹匹配失败, 则跳过叶节点的访问, 直接进入溢出缓存中继续查找 (⑥⑦). 与 APEX 和 PLIN 相比, 当键在叶节点中不存在时, PLTree 只会产生 1 个缓存行的 DRAM 访问. 而在 APEX 和 PLIN 中, 无论键是否存在于叶节点, 都必须先在叶节点中进行查找, 并产生至少 1 个 PM 块的访问. 通过设置块元数据, PLTree 可以充分利用 DRAM 高带宽特性, 减少 PM 访问次数, 合理利用有限的 PM 带宽资源.

如图 2 红色箭头所示, 当进入溢出缓存中查找键时, PLTree 首先使用 SIMD 指令在缓存元数据中逐层查找是否存在匹配的指纹 (❶❷). 如果存在指纹匹配, 则继续在溢出缓存中对应的 entry 中从右至左比较指纹匹配的键 (❸), 若键相同直接返回结果, 无需继续向左和向下层继续查找, 因为后面的即使存在相同的键也是旧数据. 当键存在时, 溢出缓存中查找操作最少只需进行一个缓存行的 DRAM 读取和一个 PM 块访问. 当键不存在时, 溢出缓存中的查找开销为 N 个缓存行的 DRAM 访问 (N 为溢出缓存的层数), 而不会产生额外的 PM 访问.

3.3 插入更新

如图 3 红色箭头所示, 与查找操作的相同, 插入更新操作需要先在内部节点中定位键所在的叶节点以及对应的元数据 (⑧⑭⑨). 在叶节点中, 使用原地更新和日志式插入的方式将键值对写入预测的 block 中. 在此过程中, 如果溢出缓存中已经存在与待插入键重复的数据, 在后续查询操作中, 这些重复数据将会被视为过期或旧数据. PLTree 对分布不均匀的真实工作负载进行了优化: 避免了插入数据之前在溢出缓存对键进行唯一性检查. 因为在 PLTree 中, 只需确保关键字在叶节点中是唯一的, 允许溢出缓存中的键存在重复. 因此, 在插入键值对之前, 只需要在叶节点的 block 中进行键的唯一性检查, 而无需继续在溢出缓存中查找该键. 具体的做法是, 在插入键值对之前, 先在块元数据和叶节点中查找键是否存在 (⑩⑪). 如果键已存在, 则会在 block 中原地更新键值对的值, 由于值的大小为 8 B, 在写入时采用 8 B 原子写入 PM, 并使用内存屏障 SFENCE 指令[39]隔离后续的写指令, 因此程序崩溃时不会出现中间状态. 如果键在 block 中不存在, 就以追加的方式将键值对写入到 block 的空闲槽位中 (⑫), 并将键的指纹加入块元数据的指纹数组之中. 键值对采用先写入值 (8 B) 再写入键 (8 B) 的方式持久化, 使用 SFENCE 指令隔离后续的写指令, 确保系统崩溃发生在值写入和键写入之间时, 槽位中的键仍为空闲标志, 避免无效键值对的写入. 上述插入方式避免了额外的 PM 访问, 因为键的唯一性检查只在叶节点中进行.

当叶节点产生由冲突导致的溢出时, APEX 和 PLIN 每次只往溢出缓存中写入一个 16 B 大小的键值对 (写放大为 16, 一个 PM 块 256 B/键值对大小 16 B). 针对 PM 中由数据访问粒度不匹配引发的写放大问题, PLTree 进行了优化: 在叶节点 block 产生溢出时选择将整个 block 的数据批量地写入到溢出缓存中 (⑲), 当 PLTree 叶节点中某个 block 中的数据全部写入溢出缓存第 0 层的同一个 entry 时, 写放大最小可降低至 1.07 (一个 PM 块 256 B/block 中的数据 240 B). 通过批量写入数据的方式, 能够根据 PM 的特性, 减小 PM 中的写放大. 此外, 每次批量写入还可以均摊数据持久化和内存屏障指令带来的开销, 从而提升系统的性能表现.

批量写入完成后, 需要通过设置空闲标志将叶节点中 block 槽位标记为空闲, 并清除对应块元数据中的指纹数据. 在批量写入缓存和将 block 槽位设置为空闲时, 即使发生系统崩溃, 也不会影响索引的一致性. 这是因为 PLTree 支持存储关键字重复的数据, 可以分为两种情况讨论: (1) 批量写入溢出缓存时发生系统崩溃. 此时可能有一部分键值对已经被写入溢出缓存中, 在叶节点 block 中的键值对未被改变, 因此系统恢复后, 叶节点中的数据仍旧存在, 对查找操作没有影响. 写入溢出缓存中的部分键值对会被当做旧的数据, 下一次批量写入时, 键重复的旧数据会被覆盖. (2) 将叶节点 block 槽位设置为空闲时系统崩溃. 此时 block 中的键值对已经全部写入溢出缓存, 叶节点 block 中可能存在部分键值对未被置空, 由于此时 block 中的键值对已经全部写入溢出缓存, 系统恢复后, 并

不会产生数据的丢失, block 与溢出缓存中重复数据将在下一次数据从 block 批量写入溢出缓存时合并.

溢出缓存采用分层的结构来存储批量写入的数据. 为了减小写放大, 当一层中某个条目数据写满时, 该条目数据会批量移入下一层 (如图 2 中❹❺). 批量写入前会合并键重复的数据, 并删除旧数据. 批量写入完成后, 更新与条目对应的缓存元数据指纹信息. 随着数据的插入, 溢出缓存中的层数将会增加. 如果溢出缓存层数过多, 将导致查找性能和范围查找性能下降. PLTree 通过扩展叶节点来解决性能下降的问题. 然而, 频繁地调整叶节点也会带来大量的开销. 为了避免这种开销, 在叶节点插入数据时通过计算叶节点溢出缓存的平均层数 (avgLevel=levelnum/size, levelnum 为叶节点中溢出缓存层数的总和, size 为 block 数组长度) 来判断是否需要进行叶节点扩展. 只有当叶节点中溢出缓存的平均层数超过一定阈值时, 才考虑对叶节点进行扩展. 在节点扩展时会对叶节点中的键重复的数据以及标记为删除的数据进行清理以减轻冗余数据带来的存储开销.

3.4 删　除

如图 3 红色箭头所示, 删除操作以键值对插入的方式进行. 当需要删除数据时, 首先在内部节点查找定位到包含目标键的叶节点 block (⑬⑭⑮). 然后, 在该 block 中检查键是否存在: 若存在, 则将对应的值标记为已删除; 若不存在, 则将键和删除标志作为键值对插入到 block 中 (⑯). 在后续执行查找操作时, 若找到的键值对中值被标记为删除, 则直接返回查找失败. 当 block 满时, 其中标记为删除的数据会随着其他有效数据一起被批量写入溢出缓存中. 溢出缓存中标记为删除的数据会在节点扩展时被清除, 以减轻出现大量删除操作时, 溢出缓存中过多的冗余数据导致的索引性能下降和额外存储开销.

3.5 范围查找

如图 3 红色箭头所示, 范围查找首先使用查询范围的最小键执行查找操作, 定位键所在的起始叶节点 (⑰②③). 然后, 根据模型定位 block 数组中第一个存储大于等于该键的 block 位置 (⑩), 并向后查找 (⑱). 叶节点之间以链表结构相连, 如果当前叶节点的查找结束, 则会继续查找下一个叶节点, 直到链表末尾或查找范围结束.

叶节点中的范围查找从叶节点中指定的起始 block 开始顺序遍历 block 数组, 并在 block 及其关联的溢出缓存中查找符合查找范围的键值对并添加至结果集合. 当访问每个 block 时, 首先读取其中的数据, 并将查询结果加入结果集合. 然后继续进入 block 对应的溢出缓存中逐层逆序读取 entry 数组中的有效数据, 同样将符合条件的数据合并到结果集中. 访问完一个 block 及其对应溢出缓存后更新查找结果集合, 具体包括如下操作: 合并相同键的键值对, 移除标记为删除的键值对, 以及在集合大小超出查找范围时删除多余的键值对.

3.6 索引恢复

当系统重启或发生崩溃导致 DRAM 中的元数据丢失时, PLTree 采用延迟恢复策略按需重建元数据. 具体的做法是设置了全局的索引版本字段 global_v. 在恢复过程中, 首先将该字段加 1, 然后通过遍历叶节点的链表来更新指向叶节点的槽位中的字段 v. 如果索引不是正常关闭, 则需要在遍历叶节点时检查 fptr 指向的父节点槽位中字段 child 指针是否正确指向自身, 如果不正确, 则更新 child 指针. 在后续的索引查询过程中, 通过比较存储在块元数据中的索引版本字段 v 和槽位中的字段 v 是否相同, 来确定是否重建元数据.

3.7 并发操作

在并发读写过程中, 应尽量减少对 PM 的访问. 例如先前的方法[24]使用了存储在 PM 中的版本锁. 当写操作获取锁时可能遭遇阻塞并且需要频繁读取锁状态. 读操作为了确保数据的一致性, 需要两次读取锁的版本号. 这都会增加不必要的 PM 带宽消耗. 此外, 完成写操作后更新锁版本也可能带来额外的 PM 写操作. 另外, 驻留在 PM 中的锁在系统崩溃时也有可能导致树永远持有锁, 处于不一致的状态.

PLTree 采用了轻量级的乐观锁 (optimistic locking) 实现高并发的读写操作. 如图 2 左所示, 在块元数据中, 为每个 block 和相应的溢出缓存存储了 1 bit 锁 (lock 字段) 和 31 bits 版本信息 (version 字段). 写操作线程使用比较并交换 (compare and swap, CAS) 指令原子地设置目标 block 的锁. PLTree 采取了同步锁定目标 block 和溢出缓存的策略, 这样即使在系统崩溃的情况下, 也能确保障数据的完整性与一致性. 当写线程成功获取锁时, 此时只有一

个写线程可以向目标 block 和溢出缓存中写入数据. 写操作完成后, 使用 CAS 指令原子地增加块元数据中版本字段 version 并释放锁. 读线程无需获取锁, 它首先读取版本锁的快照, 并检查锁是否被并发写入线程占用. 如果是的话, 读线程会等待直到写线程释放锁. 当读操作完成后, 读线程会再次读取版本信息, 以判断在读取过程中数据是否被其他写线程修改. 如果两次读取的版本不匹配, 则会重新尝试整个操作. 锁存储在 DRAM 中, 避免了 PM 访问, 从而节约 PM 带宽, 提高了索引并发性能.

PLTree 在指向叶节点的槽位中设置了叶节点锁 lock. 当叶节点进行扩展时获取锁, 此时写入操作会受阻但读操作不受影响. 接着, 采用写时复制 (copy-on-write) 策略更新叶节点: 首先, 在内存中创建叶节点的副本, 将原叶节点中的数据排序后写入新创建的叶节点. 当新的叶节点创建完成后, 通过原子操作修改内部节点槽位中的指针 child 指向新的叶节点, 随后释放锁. 为了避免额外的 PM 访问并保证崩溃一致性, 在更新叶节点链表时采取了无日志方法. 具体做法是, 在叶节点扩展完成后, 将前一个叶节点的备用指针指向新的兄弟节点, 并使用 PM 原子操作修改其中的 flag 字段来切换指针. 如果此时系统发生崩溃, 则前一个叶节点指向的下一个兄弟节点只会是旧节点或已扩展的新节点. 若崩溃发生在切换指针之后, 可能会暴露中间状态, 出现槽位中字段 child 指向的旧叶节点的情况. 但由于新叶节点存储了指向内部节点的指针 fptr, 因此在系统恢复时, 可以通过检查并更新 fptr 指向槽位中字段 child 来确保崩溃一致性.

4 实验分析

4.1 实验设置

4.1.1 实验环境和对比方案

本文实验在 Linux 服务器上进行, 该服务器操作系统为 64 位 Ubuntu 20.04.5 LTS, 内核版本为 5.4.0. 该服务器配备 512 GB DRAM 和 4 048 GB 持久化内存, 采用 Intel(R) Xeon(R) Gold 6348 CPU @ 2.60 GHz 双处理器, 每个处理器拥有 28 个物理核心和 42 MB 三级缓存. 傲腾持久化内存以 App Direct 模式配置, 并通过创建 ext4-DAX 文件系统将其挂载到文件系统进行管理. 为避免 NUMA 效应[11], 所有线程均固定到 NUMA 节点 0, 并且只能访问该节点上的 DRAM 和 PM.

本文选择了 5 个最近的持久化内存索引进行性能对比, 包括最近的持久化学习索引 APEX 以及 4 个基于 B+树结构的持久化内存索引 (FPTree, uTree, NBTree 和 DPTree). 由于 APLI 未提供源码, PLIN 为纯 PM 架构索引且实验[24]显示 PLIN 性能要低于混合架构的 APEX. 因此本文未将它们加入对比. 与 PLTree 相同, 参加对比的所有索引均为 DRAM/PM 混合架构. 其中 FPTree, uTree, NBTree 将内部节点和元数据存储在 DRAM 中. DPTree 采用分层结构, 在 DRAM 中存储缓存树, 当缓存树的大小达到一定阈值后, 使用后台线程将 DRAM 中的缓存树与持久化树进行合并. APEX 将指纹和锁等元数据存储在 DRAM 中, 内部节点和叶节点均存储在 PM 中. 以上索引均使用 PMDK 进行持久化内存的管理, 并在都在 ADR 模式[40]下运行, 采用 CLWB 和 SFENCE 指令[39]确保数据持久化. 所有索引使用 C++实现并采用-O3 优化编译. 在索引构建时, APEX 和 PLTree 的叶节点初始数据填充率设置为 50%, 以支持之后的插入操作. 参加对比索引的其他参数均采用原论文中默认配置.

4.1.2 数据集和工作负载

为了测试持久化学习型索引在合成数据和真实数据上的性能表现, 实验中选取了 1 个合成数据集 uniform 和 4 个真实数据集 face[26], osm[26], planet[41], genome[41]. 这些数据集在现有的学习型索引研究中得到广泛使用, 每个数据集由 2 亿个非重复的 8 B 无符号整数组成. 其中 uniform 数据集由均匀分布的无符号整数组成, face 数据集采样自 Facebook 用户 ID 数据, osm 和 planet 数据集采样自公开地图 OpenStreetMap 的经纬度数据和 planet ID 数据, genome 数据集包含人类染色体中的位点对数据. 其中, 真实数据集更难以拟合, 因此相较于合成数据集, 真实数据集具有更大的挑战性, 适用于评估持久化学习型索引的性能表现.

本文采用常见的工作负载来测试索引性能, 包括只读、读写混合、只写、范围查找和删除操作. 其中, 读写混合包括读重负载 (20% 写, 80% 读)、读写平衡负载 (50% 写, 50% 读) 和写重负载 (80% 写, 20% 读). 上述工作负载

使用均匀分布从数据集中选取键, 在读/写负载中, 每个键被读取/写入的概率相同. 在进行测试之前, 先预先加载 1 亿个键值对来构建索引. 范围查找从一个随机键开始, 并扫描后续的 100 条记录. 由于 DPTree 没有实现删除操作, NBTree 没有实现范围查找操作, 所以在相应的测试中不将 DPTree 和 NBTree 加入对比.

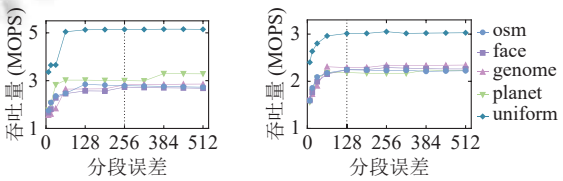
4.2 参数选择

本节将研究分段误差, 叶节点 block 大小以及溢出缓存 entry 大小的设置对索引性能的影响.

4.2.1 分段误差设置

在使用自底向上的方法构建学习索引的布局时, 分段误差的大小会直接影响键空间的划分粒度, 从而影响树的高度和 PM 访问次数. 当分段误差较小时, 键空间会被划分得更加细致, 导致更多的节点和更高的树高, 进而增加了 PM 的访问次数. 相反, 当分段误差较大时, 分区粒度过粗, 叶节点中的线性模型不能很好地拟合数据分布, 可能会引发更多的冲突, 并且更多的数据会写入溢出缓存. 因此, 在选择分段误差时需要综合考虑这两个方面的因素.

为了探究不同分段误差对 PLTree 索引性能的影响, 我们使用 5 个数据集进行了单线程实验, 并比较了不同分段误差下的查找和插入吞吐量. 实验中使用 1 亿个键构建了不同分段误差的索引, 并进行了 1 亿次查找和插入操作, 记录了操作吞吐量. 图 4 实验结果显示, 随着分段误差的增大, 所有数据集的查找和插入吞吐量逐渐上升. 当分段误差分别大于 256 (图 4(a)) 和 128 (图 4(b)) 时, 所有数据集上的查找操作和插入操作的吞吐量逐渐收敛. 这是因为随着分段误差的增大, 索引每层中的节点数量减少, 从而在遍历过程中缓存命中率提高, 同时索引高度逐步减小, 从而减少了 PM 的访问. 此外, 在查找操作中采用了存储在 DRAM 中的指纹数据加速溢出缓存的查找, 溢出缓存的每一层只会产生一个缓存行的 DRAM 访问, 当溢出缓存层中不存在键时则跳过该层 PM 访问, 每次在溢出缓存查找最多只会产生一次 PM 访问, 因此索引查找性能受到溢出缓存的影响较小. 对于插入操作, 由于只需在叶节点数据块中原地更新和插入键值对, 只有在数据块满时, 才会将数据批量写入溢出缓存, 避免了溢出缓存的频繁写入, 插入操作的性能主要受索引高度的影响. 当误差继续增大时索引的高度不再增加, 因此随着分段误差增大读写性能逐渐收敛. 根据实验结果, 我们选择将索引构建的分段误差大小设置为 256.



(a) 查找操作吞吐量 (b) 插入操作吞吐量

图 4 使用不同分段误差构建的索引在不同数据集上进行查找和插入操作时的吞吐量

4.2.2 叶节点中 block 大小和溢出缓存中 entry 大小对性能的影响

为了研究叶节点中 block 和溢出缓存中 entry 的大小设置对索引性能的影响, 我们在 4 个真实数据集上进行了单线程性能测试. 比较了数据结构在不同大小设置下 (同时修改元数据大小以匹配 block 和 entry 的不同大小设置), 进行 1 亿次查找和插入操作时的吞吐量. 根据图 5(a) 和图 5(b) 的结果, 我们发现将叶节点 block 大小设置为 256 B 时, 可以获得最高的读写吞吐量. 由于真实数据集键空间的局部难以拟合, 较小的 block 更容易因冲突导致数据频繁写入溢出缓存, 从而增加查找路径长度及查找时间. 对于插入操作, 较小的 block 更加容易被写满引起频繁的溢出缓存写入, 降低了性能. 当 block 大小超过 256 B 时, 读写性能开始下降. 这是因为块元数据需存储更多指纹信息, 可能导致 DRAM 缓存未命中的概率增加. 另外, 由于指纹大小为 8 bits, 不同键存在相同指纹的概率为 1/256, 因此当 block 增大时, 可能出现不同的键具有相同的指纹信息, 导致读写操作可能需要在叶节点访问不止一个 PM 块来查找目标键.

根据图 5(c) 和图 5(d) 的结果, 我们发现将溢出缓存中 entry 的大小设置为 768 B 时, 大部分的数据集上索引的读写吞吐量达到最高点. 当 entry 较小时, 由于溢出缓存每一层的数据容纳能力有限, 溢出缓存的层数较多, 在查找键时需要访问更多的缓存行, 增加了缓存未命中的概率. 同样, 在插入操作时, 若 entry 较小可能会导致数据频繁

地被写入下一层, 从而降低吞吐量. 另一方面, 当 entry 过大时, 缓存元数据需要维护更多的指纹信息. 这会增加在访问溢出缓存时缓存未命中的概率. 因此根据实验结果, 我们将 block 大小设置为 256 B, entry 大小设置 768 B.

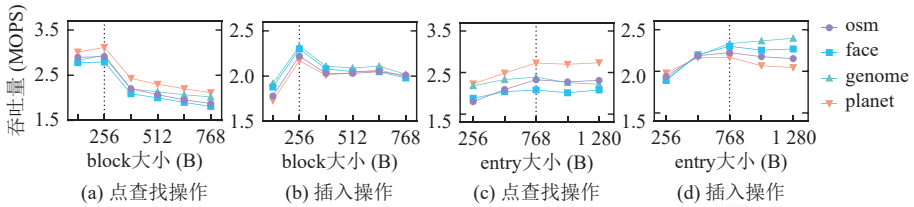


图 5 在不同 block 大小和不同 entry 大小设置下进行查找和插入操作的吞吐量

4.3 单线程测试

本节中, 我们对索引进行了单线程实验, 并且使用 5 个不同的数据集分别进行了 1 亿次的点查找/插入/范围查找/删除操作的性能测试. 如图 6(a) 所示 PLTree 和 APEX 的点查找在所有的数据集上的性能均优于 B+树结构的索引, 这是因为学习索引根据数据的分布特点构建索引, 能够创建更大的数据节点, 因此具有更扁平的索引结构, 而传统 B+树的结构的索引构建与数据集的分布无关, 叶节点的数量和树的高度随着数据量的增加而增加, 因此查找遍历时会产生更多的缓存未命中. 相比于其他 4 个真实数据集, PLTree 和 APEX 在合成数据集 uniform 上所有操作的吞吐量最高, 这是因为在该数据集上键分布均匀且易于模型拟合, 索引高度最低, 此外在叶节点中所产生的冲突也是最小的, 大部分的键值对存储在叶节点中, 无需频繁访问溢出缓存结构.

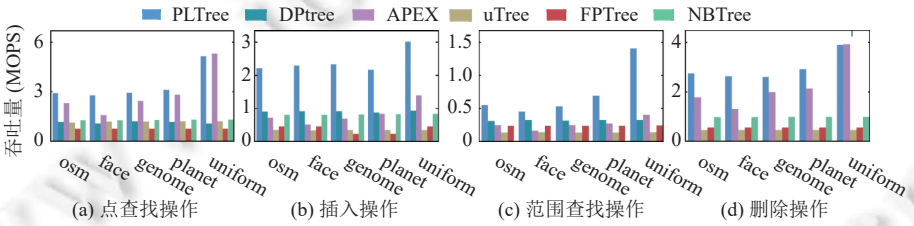


图 6 在 5 个数据集上进行单线程的点查找/插入/范围查找/删除的吞吐量

如图 6(a) 和图 6(d) 所示, 在 4 个真实数据集上进行点查找和删除操作时, PLTree 相比其他索引结构表现更优异. 以线性模型较难拟合的 face 数据集为例, PLTree 点查找性能分别比 DPtree/APEX/uTree/FPTree/NBTree 提升了 2.6 倍/1.8 倍/2.3 倍/3.7 倍/2.2 倍. PLTree 在性能上优于 APEX 的原因是, 真实数据集中的键呈非线性特征, 导致学习型索引中有较多数据需要存储在溢出缓存中. 而 PLTree 通过块元数据结构提高了查找性能, 可以在访问叶节点之前先通过访问块元数据判断叶节中键是否存在, 若不存在则可直接跳过叶节点的访问, 从而避免了额外的 PM 访问. 相比之下, APEX 使用的 probe-and-stash 技术需要先在叶节点中进行线性探测来判断键是否存在, 然后才会在溢出缓存 (stash) 中查找键, 从而产生了不必要的 PM 访问. 此外, 在查找溢出缓存中的键时, PLTree 的缓存元数据结构中每个缓存行可以存储 48 个指纹信息, 而 APEX 的元数据结构中每 2 个缓存行只存储了 15 个指纹信息, 因此 APEX 比 PLTree 更容易产生更多的缓存行访问. 在合成数据集 uniform 上 PLTree 的删除操作的吞吐量与 APEX 相当, 点查询的吞吐量接近于 APEX, 这是因为 uniform 数据集中的数据是线性分布的, 大部分的数据都存储在叶节点中, PLTree 在访问叶节点之前需要额外先访问块元数据, 产生了一定的性能损失.

如图 6(b) 和图 6(c) 所示, PLTree 在所有数据集上的插入和范围查找性能都优于其他索引. 以插入操作为例, 在 uniform 数据集上, 相对于 DPtree/APEX/uTree/FPTree/NBTree, PLTree 的性能提高了 3.2 倍/2.2 倍/8.4 倍/6.5 倍/3.6 倍. 这是因为: (1) PLTree 避免了在溢出缓存中进行键的唯一性检查, 从而缩短了插入操作的查找路径; (2) 每次插入和更新操作只发生在叶节点, 减少了冲突严重时将新插入的数据写入溢出缓存的频率; (3) PLTree 采用了写优化的溢出缓存结构, 当叶节点 block 满时, 采用批量写入的方式将数据移入下一层, 从而减小了 PM 的写放大.

此外, 可以发现在 4 个真实数据集上, APEX 的插入操作性能较传统结构索引 DPTree 和 NBTree 弱, 主要是因为 APEX 面对真实数据集时叶节的冲突增加, APEX 在插入操作时需要进行额外的 PM 和缓存访问, 以检查键在叶节点和 stash 中是否已经存在. 并且每次向 stash 插入数据时只写入 16 B 数据, 导致 16 倍的 PM 写放大. 而 DPTree 采用了分层的结构, 新插入的数据先写入 DRAM 驻留的缓冲树中, 之后再通过后台线程批量写入持久化树中, 从而提高了写入性能. 对于范围查找, PLTree 在所有数据集上性能均优于其他索引, 在 uniform 数据集上 PLTree 相对于 DPtree/APEX/uTree/FPTree 性能分别提高了 4.3 倍/3.5 倍/10.4 倍/5.8 倍. 这是因为 PLTree 相较于 B+树结构持久索引具有更大的叶节点, 范围查找时可以在连续的地址范围内顺序访问 PM, 从而具有更高 PM 带宽利用率. 而 DPTree 需要搜索多个树并组合结果, 影响了范围查找性能. uTree 的范围查找性能最低, 因为 uTree 将每个记录都存储在链表节点中, 遍历它们会导致许多缓存未命中.

4.4 多线程测试

本节的实验测试了索引在不同负载上的多线程可扩展性, 实验中线程从 1 个线程扩展到 28 个线程, 所有的线程被固定在物理核心上运行, 每个线程独立使用一个物理核心.

图 7–图 10 展示了单个操作的并发性能, 包括 100% 的点查询/插入/范围查询/删除操作. 实验结果表明, 在不同的负载下, PLTree 在所有的数据集上都表现出了良好的多线程可扩展性, 随着线程数量增加, PLTree 的吞吐量也相应地增加, 且增加的趋势呈现线性或近似线性. 在 4 个真实数据集上, PLTree 的单项操作负载的并发性能均优于其他索引, 这是因为: 首先, PLTree 通过拟合数据集的分布, 采用基于模型的搜索, 提高了 CPU 缓存利用率; 其次, PLTree 采用轻量级并发控制, 减少了同步开销; 第三, PLTree 利用 DRAM 中的元数据来加速查找, 避免了不必要的 PM 访问, 从而减少了有限的 PM 带宽争用, 保证了高性能; 最后, PLTree 采用了批量写入和追加写入的方式将数据写入溢出缓存, 降低了 PM 的写放大, 提高了 PM 带宽的利用率. 另外与单线程实验的情况一样, 在 uniform 数据集上, PLTree 的点查询和删除操作并发性与 APEX 性能相近.

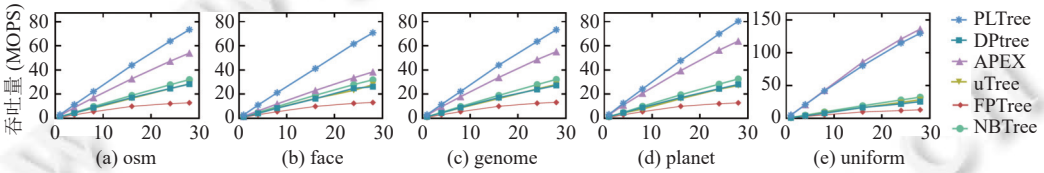


图 7 多线程点查询操作吞吐量, 横坐标为线程数, 纵坐标为吞吐量

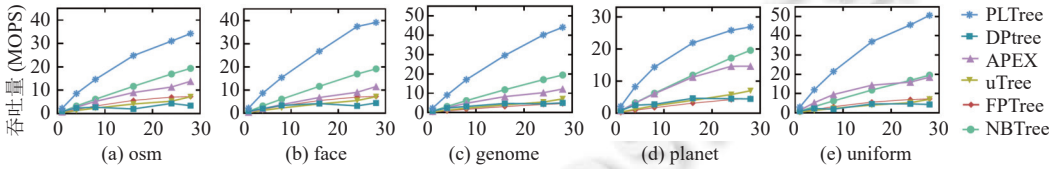


图 8 多线程插入操作吞吐量, 横坐标为线程数, 纵坐标为吞吐量

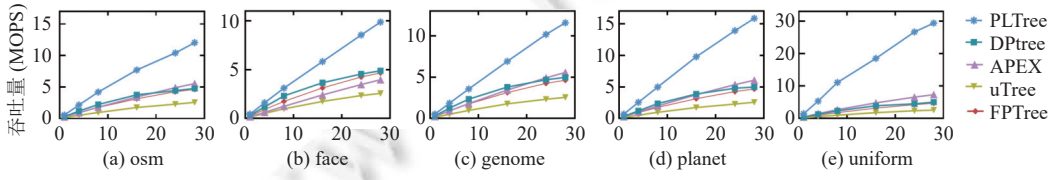


图 9 多线程范围查询操作吞吐量, 横坐标为线程数, 纵坐标为吞吐量

图 11–图 13 展示了混合负载的并发性能. 实验结果表明, 混合负载测试中, PLTree 在所有的数据集上展现了最佳的性能和可扩展性. 随着负载中插入操作比例的增加, 其他的索引的扩展性变差, 而 PLTree 仍然保持最

佳的性能. 这得益于 PLTree 写入优化的溢出缓存以及 DRAM 元数据的合理利用, 避免不必要的 PM 访问, 减小了 PM 带宽的争用, 降低了同步的开销. 另外, 实验结果显示 APEX 在真实数据集上的混合负载性能表现不佳, 如图 11(b)、图 12(a)–图 12(b)、图 13(a)–图 13(d) 所示, 在 osm 和 face 数据集上, APEX 的混合负载吞吐量要低于 NBTree, 这是因为 osm 和 face 数据集非线性程度较高、较难拟合, APEX 叶节点中冲突较多, 大量的数据被插入溢出缓存 stash 中导致了 PM 写放大. 此外, 频繁访问 stash 还降低了 CPU 缓存利用率, 抵消了学习索引模型搜索带来的优势.

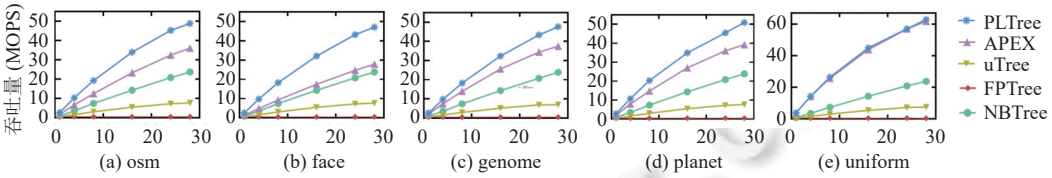


图 10 多线程删除操作吞吐量, 横坐标为线程数, 纵坐标为吞吐量

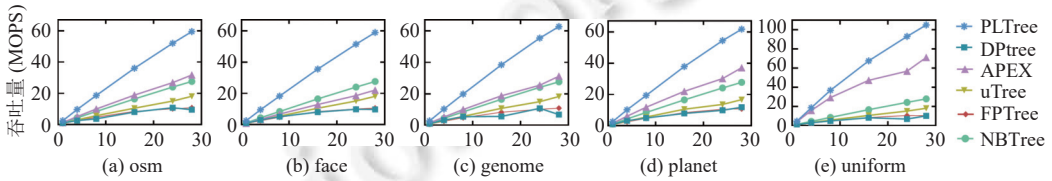


图 11 多线程读重负载吞吐量, 横坐标为线程数, 纵坐标为吞吐量

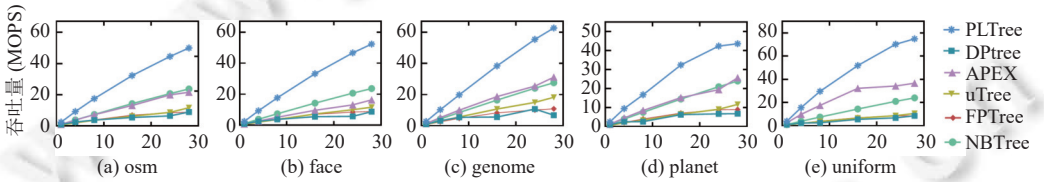


图 12 多线程读写平衡负载吞吐量, 横坐标为线程数, 纵坐标为吞吐量

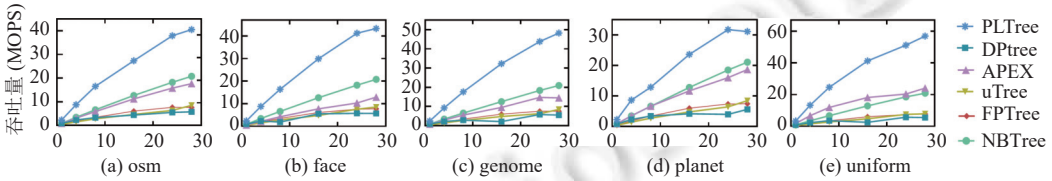


图 13 多线程写重负载吞吐量, 横坐标为线程数, 纵坐标为吞吐量

图 14 展示了在 28 线程下索引运行倾斜负载的性能. 实验使用 4 个真实数据集在不同的 zipfian 倾斜度 (0.2–0.99) 下进行了 1 亿次点查询的吞吐量测试. 实验结果表明, 除了 FPTree 之外, 其他索引在倾斜度增加时吞吐量都表现出增高的趋势, 且 PLTree 在不同倾斜度下的查找性能优于其他索引. 较高的倾斜度意味着只有一小部分热门键被频繁访问, 而大多数键很少被访问. 这说明 PLTree/DPtree/APEX/uTree/NBTree 能够有效利用 CPU 缓存来加速热点数据的访问. 通过横向对比 4 个数据集中每个索引的吞吐量 (图 14(a)–图 14(d)), 我们可以发现 APEX 的性能容易受到不同数据集分布的影响, 而 PLTree 和 B+树结构的索引查询吞吐量的变化不大. 这说明 PLTree 的设计能够很好地适应不同的访问模式, 并且性能受数据集分布的影响较小.

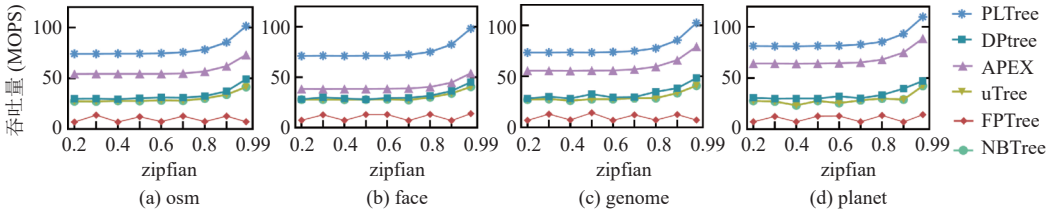


图 14　在 28 线程不同倾斜负载设置下 4 个真实数据集上的查找操作吞吐量

4.5　索引加载时间

本节实验对 6 个索引的加载时间进行了评估, 我们使用了 1 亿个键来构建索引, 并记录了构建索引所需的时间. 由于 DPtree、FPTree、uTree 和 NBTree 没有实现批量加载功能, 因此我们使用它们的插入函数来进行索引构建. 图 15 显示了在 6 个数据集上构建索引所需的时间. 实验结果表明, 在所有数据集上, PLTree 的加载时间最短 (5.3–12.6 s), 其次是 APEX (8.6–26.2 s). 相比之下, 传统的 B+树结构索引则需要更长加载时间, 平均约 94–324 s. 由于 PLTree 采用了 O(n) 复杂度的分段算法来构建索引, 因此具有最快的加载速度. 即使在较难拟合的 face 数据集上只需约 12 s, DPtree/APEX/uTree/FPTree/NBTree 的加载时间分别是 PLTree 加载时间的 7.4 倍/2.0 倍/26.6 倍/13.0 倍/8.5 倍, 进一步证明了 PLTree 在索引构建方面表现出的高性能.

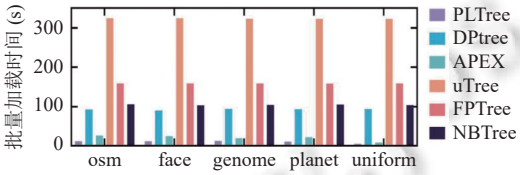


图 15　不同数据集上索引的加载时间

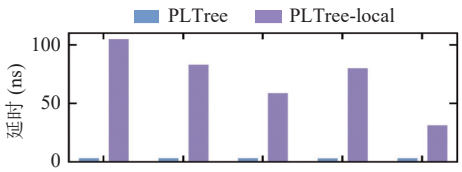


图 16　不同数据集上内部节点查找的平均延时

此外, 实验结果还显示 PLTree 和 APEX 的索引加载时间受数据分布的影响. 数据集的非线性程度越高, 索引的加载时间越长. 例如, 线性分布的 uniform 数据集所需的加载时间最短 (PLTree 为 5.3 s, APEX 为 8.6 s). 而在真实数据集上, APEX 相比 PLTree 的加载所需时间更长. 在 osm 数据集上, APEX 的索引加载时间约为 PLTree 加载时间的 2.1 倍. 这是因为 APEX 在加载键时需要同时计算成本模型估计索引的查找/插入平均延时, 并选择合适的分割数量, 从而导致更多的计算开销. 因此, APEX 在加载真实数据集时比 PLTree 需要更多时间.

4.6　索引设计对性能影响

4.6.1　两阶段构建索引

PLTree 采用了两阶段的方法构建, 通过消除内部节点中的局部查找来提高性能. 我们首先对比了表 1 中所示的 5 个索引, 在不同数据集上进行 1 亿次查询时, 内部节点遍历的平均访问延时 (由于 DPTree 采用了分层树的结构, 所以没有被纳入比较范围).

表 1　内部节点平均查找延时 (ns)

数据集	PLTree	APEX	uTree	FPTree	NBTree
osm	**3.03**	26.60	482.67	853.44	315.68
face	**3.01**	17.71	483.70	852.56	315.92
genome	**3.01**	9.10	486.01	854.58	315.32
planet	**3.00**	13.84	482.19	858.37	315.81
uniform	**3.03**	6.73	481.69	848.67	315.87

从表 1 的结果来看, PLTree 在所有测试数据集上的平均查找延时均低于其他索引, 最低仅为 3 ns. 并且内部节点遍历的平均延时几乎不受数据分布的影响. 原因在于 PLTree 消除了内部节点中的局部查找, 每层只需访问一

个 PM 块的. 同时, PLTree 是一棵平衡树, 所以查找所有内部节点时的遍历路径长度相同. 由于 PLTree 内部节点的高度较扁平, 各个数据集上的内部节点高度相差不大, 因此数据分布对内部节点的查找性能负面影响较小. 相比之下, APEX 的内部节点平均访问延时约为 PLTree 的 2.2–8.8 倍, 并且内部节点的访问延时受到数据分布的影响. 这是因为 APEX 是一棵非平衡树, 构建索引时内部节点的高度受到键局部分布影响. 尽管 APEX 内部节点中无需进行局部搜索, 但当键的局部分布难以拟合时, 内部节点的遍历路径增加将影响内部节点的查找性能.

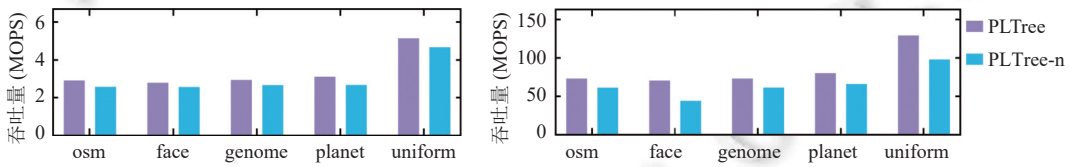
B+树类持久化内存索引的内部节点遍历平均延时最长, 约为 PLTree 的 105–283 倍. 虽然它们都将内部节点存储在 DRAM 中以加速访问, 但与学习型索引相比, B+树类结构的索引内部节点遍历路径更长, 并且需要在内部节点中进行局部搜索来查找子节点的指针, 这导致了更多的 CPU 缓存行未命中, 从而影响了查找性能.

为了验证消除内部节点的局部搜索是否能够提升性能, 我们设计了消融实验: 在构建索引时, 保留了内部节点的局部搜索. 叶节点和内部节点的划分误差与原索引相同, 都设置为 256. 在内部节点中使用二分法进行局部搜索, 其他数据结构的设计与原索引保持一致. 经过 1 亿次查找后, 记录了内部节点的平均读取延时. 如图 16 所示, PLTree-local 代表经过修改后的索引, 即保留了内部节点的局部搜索. 实验结果显示, 消除内部节点的局部搜索后, PLTree 的内部节点性能显著提升. 在 osm/face/genome/planet/uniform 数据集上, 查找性能分别提升了 34.7 倍/27.6 倍/19.5 倍/26.6 倍/10.3 倍.

此外, 我们还观察到, PLTree-local 内部节点的查找性能受到数据分布的影响, 在难以拟合的数据集中, 内部节点的查找延时更长. 以 osm 数据集为例, PLTree-local 内部节点的平均延时约为 105 ns, 而 PLTree 不会受数据分布影响, 在所有的数据集上内部节点查找平均仅需 3 ns. 这说明通过两阶段索引构建消除内部节点中的局部搜索能够在提升性能的同时消除数据分布对索引内部节点查找性能的负面影响.

4.6.2 块元数据的影响

本节分析了元数据结构对性能的影响. 为了进行对比, 我们在关闭块元数据加速后, 分别在单线程和 28 线程环境下执行了 1 亿次查找操作. 如图 17 所示, PLTree-n 代表关闭了块元数据加速的索引. 通过观察吞吐量, 我们发现关闭块元数据加速后, 索引的性能下降. 这是因为关闭块元数据加速后, 无论查找的目标键在叶节点中是否存在, 都必须先对叶节点进行访问. 而对于冲突较大的叶节点, 这会产生更多额外的 PM 访问. 然而, 当使了块元数据加速后, 如果叶节点中不存在键, 只需进行一个缓存行的 DRAM 访问, 跳过了叶节点的访问. 这避免了 PM 带宽的消耗, 并且在多线程情况下, 还能避免有限的 PM 带宽争用, 提高了吞吐量.

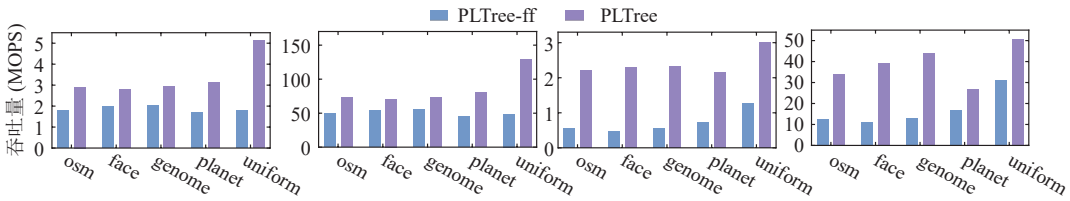


(a) 单线程查找操作吞吐量　　(b) 28 线程查找操作吞吐量

图 17　对比关闭块元数据加速时在不同数据集上查找操作的吞吐量

4.6.3 溢出缓存的影响

为了评估溢出缓存结构设计的有效性, 我们设计了对比实验, PLTree-ff 使用树结构的持久化内存索引 Fast&Fair 替代文中提出的溢出缓存, 并在单线程和 28 线程环境下分别测试了 1 亿次查找和 1 亿次插入操作的吞吐量. 如图 18 所示, 实验结果表明, 采用本文中提出的溢出缓存能够有效提升索引的查找和插入性能. PLTree 溢出缓存采用缓存元数据加速了查找, 使得当键存在于溢出缓存时, 最少仅需读取一个 PM 块即可完成查找. 而在 Fast&Fair 中查找键时, 可能需要访问多个 PM 块. 插入过程中, 为确保键的唯一性, PLTree-ff 必须先查找 Fast&Fair 以避免重复的键, 而 PLTree 避免了在溢出缓存中对键进行唯一性检查, 从而在插入数据前减少了不必要的 PM 访问, 节约了 PM 带宽. 此外, 在 Fast&Fair 叶节点插入键值对时, 会产生写放大, 而 PLTree 溢出缓存采用日志形式批量写入数据, 避免了 PM 的写放大, 提高了 PM 带宽利用率.



(a) 查找操作 (1 线程) (b) 查找操作 (28 线程) (c) 插入操作 (1 线程) (d) 插入操作 (28 线程)

图 18 对比不同溢出缓存设计在不同数据集上查找操作和插入操作的吞吐量

4.7 空间开销测试

在本节中我们首先对比了两个持久化学习索引 PLTree 和 APEX 分别在合成数据集和真实数据集上写入 2 亿个键值对后的 DRAM 和 PM 的空间开销, 实验结果显示在合成数据集上 PLTree 的存储空间开销要比 APEX 低. 在真实数据集上 PLTree 的 DRAM 空间开销比 APEX 低, 但是 PM 空间开销要比 APEX 要高, 详见表 2.

表 2 PLTree 和 APEX 存储开销 (GB)

索引	uniform (DRAM/PM)	face (DRAM/PM)
PLTree	0.495/4.391	0.859/8.934
APEX	0.571/5.379	1.544/5.494

为进一步探究 PLTree 中重复数据写入时存储空间的消耗情况, 我们在真实数据集 face 上构建了一个包含 200M 个 (即 2 亿个) 键值对的索引, 并进行了不同比例的重复数据随机插入实验. 如表 3 所示, 我们分别向索引随机插入原数据集中 10% 至 50% 的键重复的数据, 即分别对应了额外插入 20M/40M/60M/80M/100M 个重复键值对. 表 3 的实验结果显示, 随着重复数据写入比例的增加, PLTree 在 PM 上的空间开销相对于原始状态分别增长了约 4.17%、6.91%、9.33%、11.66% 和 13.95%, 而在 DRAM 中的空间开销也相应增加了约 3.51%、5.82%、7.87%、9.85% 及 11.81%. 每增加 10% 的重复数据, PM 和 DRAM 空间开销的增量平均约为 2.45% 和 2.08%.

表 3 不同比例重复数据写入 PLTree 时存储空间的消耗情况

存储空间开销增量	10%	20%	30%	40%	50%
PM空间开销增量 (%)	4.17	6.91	9.33	11.66	13.95
DRAM空间开销增量 (%)	3.51	5.82	7.87	9.85	11.81

以上实验结果表明, 在真实数据分布下, PLTree 牺牲了一定的空间存储空间以换取更高的读写性能, 本文提出的索引设计方案能够一定程度上减轻由于冗余数据导致的空间开销增大问题.

4.8 索引信息统计分析

本节首先分析了加载 1 亿条数据时, PLTree 和 APEX 在不同数据集上的索引高度和叶节点数, 结果如表 4 所示. PLTree 在大部分数据集上能够构建较少的叶节点, 并且是一棵平衡树, 索引的最大最小高度相同. APEX 是一颗非平衡树, 在难以拟合的局部键空间可能需要构建更长的遍历路径, 因此查询性能更易受数据分布的影响.

表 4 PLTree 和 APEX 索引高度和叶节点数统计 (最小高度/最大高度/叶节点数)

索引	osm	face	genome	planet	uniform
PLTree	4/4/77 941	3/3/126 140	3/3/21 007	4/4/51 814	3/3/683
APEX	2/6/80 524	2/6/113 839	2/6/22 485	2/4/59 713	3/4/16 384

表 5 展示了在索引加载 1 亿个数据时, 不同数据集上 block 总数、溢出缓存总数以及其占 block 总数的比例, 结果显示在 osm/face/genome/planet/uniform 上溢出缓存占比分别 11.87%/12.22%/16.64%/11.83%/0.001 3‰. 这表明溢出缓存的数量与数据的分布特点有关, 分布越均匀, 溢出缓存的数据量越少. 在此基础上, 我们进一步统计了

PLTree 进在行 1 亿次查询时, 每次查询过程中访问元数据所需的平均缓存行数, 以及内部节点、叶节点和溢出缓存对应的 PM 块的平均访问数. 在所有数据集上元数据的平均仅产生约 1 个缓存行的访问, 内部节点 PM 块平均访问数与内部节点的高度有关, 每一层产生 1 个 PM 块的访问. 除去内部节点的 PM 访问, 在 uniform 数据集上, PM 的访问主要集中在叶节点, 平均约为 1 个 PM 块. 在真实数据集上, 叶节点和溢出缓存的 PM 块平均访问数之和接近 1 个 PM 块, 这说明 PLTree 的索引设计在面对真实数据集时, 能够有效地减少不必要的 PM 访问, 充分利用模型搜索的优势, 使索引的在真实数据集上的 PM 块访问数尽量接近在合成数据集上的访问数.

表 5 PLTree 索引信息统计

统计信息	osm	face	genome	planet	uniform
block数	13 386 188	13 400 613	13 344 913	13 360 928	13 333 695
溢出缓存数	1 588 894	1 638 007	2 220 666	1 580 603	17
溢出缓存数/block数	11.87%	12.22%	16.64%	11.83%	0.001 3‰
元数据平均访问数 (缓存行)	1.080	1.114	1.025	1.002	1.000
内部节点PM块平均访问数	3	2	2	3	2
叶节点PM块平均访问数	0.566	0.428	0.385	0.670	1.000
溢出缓存PM块平均访问数	0.504	0.659	0.693	0.372	3.370×10^{-6}

4.9 索引恢复性能测试

在本节中, 我们在不同索引中分别加载 1 百万至 1 亿个键值对, 模拟索引崩溃并进行恢复操作, 评估各索引的恢复速度. NBTree 没有提供索引恢复的实现, 故未参与此次对比. PLTree 和 APEX 采用了延迟恢复策略. 因此, 针对这两者的实验包含了数据恢复和性能恢复两部分. 其中性能恢复时间为索引执行查询时性能恢复稳定的时间. 表 6 展示了在 planet 数据集上索引的恢复性能, PLTree/uTree/FPtree/DPTree 的索引恢复时间随着数据量的增加而增加, 这是因为它们需要遍历叶节点来恢复索引. uTree/FPtree/DPTree 在恢复时需要在内存中重建内部节点, 增加了恢复时间. 其中 uTree 在恢复时需要遍历键值对组成的链表, 缓存利用率较低, 恢复时间最长. 而 PLTree 通过学习数据的分布可以构建更大的数据节点, 叶节点数较少, 且恢复时无需重建内部节点, 因此可以获得较快的索引恢复速度. APEX 在恢复时只需读取日志并重做或撤销崩溃时进行的结构修改操作, 因此索引恢复时间和数据量无关. 另外 APEX 的性能恢复时间要比 PLTree 快, 这是因为与 APEX 相比 PLTree 在性能恢复时需要重建叶节点中的块元数据, 因此牺牲了一定的性能恢复时间.

表 6 索引恢复时间 (s)

键值对数量	PLTree (索引恢复/性能恢复)	APEX (索引恢复/性能恢复)	uTree	FPTree	DPTree
1M	0.000 12/0.132	0.025 9/0.129	0.170	0.019	0.018
10M	0.001 85/0.621	0.027 5/0.495	1.688	0.229	0.035
50M	0.011 4/2.042	0.027 9/1.870	8.235	1.226	0.151
100M	0.025 2/4.395	0.027 3/2.279	16.379	2.479	0.295

5 总 结

本文提出了一种 DRAM/PM 混合架构的持久化学习索引 PLTree, 优化了真实数据负载下的读写性能. PLTree 利用模型搜索来提高在 PM 上的查找性能, 并采用了两阶段的索引构建方法, 通过消除学习索引内部节点中的局部搜索, 减少 PM 的访问次数, 同时减轻了数据分布对索引访问性能的影响. 根据学习索引面对真实数据集时的特点, 合理利用 DRAM 中的元数据加速持久化学习索引查询, 避免了额外的 PM 访问. 针对持久化内存的特性, 设计了写入优化的分层溢出缓存, 存储叶节点中由于冲突导致的溢出数据, 提升了在真实数据负载中索引的写入性能. 实验结果显示 PLTree 在处理真实数据时, 牺牲了一定的存储空间以换取更高的读写性能. 本文提出索引设计能够在一定程度减轻这一问题, 但是在面对更大的数据规模时可能会导致一定的空间开销增长. 作为未来的优化方向,

我们将继续探索如何更有效地降低 PLTree 的空间开销, 力求在空间利用和性能之间取得更好的平衡, 并进一步探讨优化并发控制机制, 以期在提升并发性能的同时, 保证数据的一致性与安全性.

References:

[1] Lu BT, Ding JL, Lo E, Minhas UF, Wang TZ. APEX: A high-performance learned index on persistent memory. Proc. of the VLDB Endowment, 2021, 15(3): 597–610. [doi: 10.14778/3494124.3494141]

[2] 3D XPoint™: A breakthrough in non-volatile memory technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html

[3] Lersch L, Hao XP, Oukid I, Wang TZ, Willhalm T. Evaluating persistent memory range indexes. Proc. of the VLDB Endowment, 2019, 13(4): 574–587. [doi: 10.14778/3372716.3372728]

[4] Hwang D, Kim WH, Won Y, Nam B. Endurable transient inconsistency in Byte-addressable persistent B+-tree. In: Proc. of the 16th USENIX Conf. on File and Storage Technologies. Oakland: USENIX Association, 2018. 187–200.

[5] Yang J, Wei QS, Chen C, Wang CD, Yong KL, He BS. NV-Tree: Reducing consistency cost for NVM-based single level systems. In: Proc. of the 13th USENIX Conf. on File and Storage Technologies. Santa Clara: USENIX Association, 2015. 167–181.

[6] Oukid I, Lasperas J, Nica A, Willhalm T, Lehner W. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In: Proc. of the 2016 Int'l Conf. on Management of Data. San Francisco: ACM, 2016. 371–386. [doi: 10.1145/2882903.2915251]

[7] Liu JH, Chen SN, Wang LJ. LB+trees: Optimizing persistent index performance on 3DXPoint memory. Proc. of the VLDB Endowment, 2020, 13(7): 1078–1090. [doi: 10.14778/3384345.3384355]

[8] Zhang BW, Zheng S, Qi ZL, Huang LP. NBTree: A lock-free pm-friendly persistent B+-tree for eADR-enabled PM systems. Proc. of the VLDB Endowment, 2022, 15(6): 1187–1200. [doi: 10.14778/3514061.3514066]

[9] Lee SK, Lim KH, Song H, Nam B, Noh SH. WORT: Write optimal radix tree for persistent memory storage systems. In: Proc. of the 15th USENIX Conf. on File and Storage Technologies. Santa Clara: USENIX Association, 2017. 257–270.

[10] Ma SN, Chen K, Chen SM, Liu MX, Zhu JL, Kang HB, Wu YW. ROART: Range-query optimized persistent art. In: Proc. of the 19th USENIX Conf. on File and Storage Technologies. USENIX Association, 2021. 1–16.

[11] Kim WH, Krishnan RM, Fu XW, Kashyap S, Min C. PACTree: A high performance persistent range index using pac guidelines. In: Proc. of the 28th ACM SIGOPS ACM Symp. on Operating Systems Principles. Virtual Event: ACM, 2021. 424–439. [doi: 10.1145/3477132.3483589]

[12] Zhou XJ, Shou LD, Chen K, Hu W, Chen G. DPTree: Differential indexing for persistent memory. Proc. of the VLDB Endowment, 2019, 13(4): 421–434. [doi: 10.14778/3372716.3372717]

[13] Chen YM, Lu YY, Fang KD, Wang Q, Shu JW. uTree: A persistent B+-tree with low tail latency. Proc. of the VLDB Endowment, 2020, 13(12): 2634–2648. [doi: 10.14778/3407790.3407850]

[14] Arulraj J, Levandoski J, Minhas UF, Larson PA. BzTree: A high-performance latch-free range index for non-volatile memory. Proc. of the VLDB Endowment, 2018, 11(5): 553–565. [doi: 10.1145/3187009.3164147]

[15] Debnath B, Haghdoost A, Kadav A, Khatib MG, Ungureanu C. Revisiting hash table design for phase change memory. ACM SIGOPS Operating Systems Review, 2016, 49(2): 18–26. [doi: 10.1145/2883591.2883597]

[16] Zuo PF, Hua Y, Wu J. Level hashing: A high-performance and flexible-resizing persistent hashing index structure. ACM Trans. on Storage, 2019, 15(2): 13. [doi: 10.1145/3322096]

[17] Nam M, Cha H, Choi YR, Noh SH, Nam B. Write-optimized dynamic hashing for persistent memory. In: Proc. of the 17th USENIX Conf. on File and Storage Technologies. Boston: USENIX Association, 2019. 31–44.

[18] Lu BT, Hao XP, Wang TZ, Lo E. Dash: Scalable hashing on persistent memory. Proc. of the VLDB Endowment, 2020, 13(8): 1147–1161. [doi: 10.14778/3389133.3389134]

[19] Hu DK, Chen ZW, Che WK, Sun JH, Chen H. Halo: A hybrid PMEM-DRAM persistent hash index with fast recovery. In: Proc. of the 2022 Int'l Conf. on Management of Data. Philadelphia: ACM, 2022. 1049–1063. [doi: 10.1145/3514221.3517884]

[20] Vogel L, van Renen A, Imamura S, Giceva J, Neumann T, Kemper A. Plush: A write-optimized persistent log-structured hash-table. Proc. of the VLDB Endowment, 2022, 15(11): 2895–2907. [doi: 10.14778/3551793.3551839]

[21] Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N. The case for learned index structures. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 489–504. [doi: 10.1145/3183713.3196909]

[22] Zhang Z, Jin PQ, Xie XK. Learned indexes: Current situations and research prospects. Ruan Jian Xue Bao/Journal of Software, 2021,

32(4): 1129–1150 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6168.htm [doi: 10.13328/j.cnki.jos.006168]

[23] Chen LY, Chen SM. How does updatable learned index perform on non-volatile main memory? In: Proc. of the 37th IEEE Int'l Conf. on Data Engineering Workshops. Chania: IEEE, 2021. 66–71. [doi: 10.1109/ICDEW53142.2021.00019]

[24] Zhang Z, Chu ZL, Jin PQ, Luo YP, Xie XK, Wan SH, Luo Y, Wu XF, Zou P, Zheng CY, Wu GA, Rudoff A. PLIN: A persistent learned index for non-volatile memory with high performance and instant recovery. Proc. of the VLDB Endowment, 2022, 16(2): 243–255. [doi: 10.14778/3565816.3565826]

[25] Wang ZH, Lai BL, Zhao ZY, Lu K, Wan JG. APLI: A high-performance learned index for persistent memory. Journal of Chinese Computer Systems, 2024, 45(9): 2110–2118 (in Chinese with English abstract). [doi: 10.20009/j.cnki.21-1106/TP.2023-0140]

[26] Kipf A, Marcus R, Van Renen A, Stoian M, Kemper A, Kraska T, Neumann T. SOSD: A benchmark for learned indexes. arXiv:1911.13014, 2019.

[27] Xie Q, Pang CY, Zhou XF, Zhang XL, Deng K. Maximum error-bounded piecewise linear representation for online stream approximation. The VLDB Journal, 2014, 23(6): 915–937. [doi: 10.1007/s00778-014-0355-0]

[28] Ding JL, Minhas UF, Yu J, Wang C, Do J, Li YN, Zhang HT, Chandramouli B, Gehrke J, Kossmann D, Lomet D, Kraska T. ALEX: An updatable adaptive learned index. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 969–984. [doi: 10.1145/3318464.3389711]

[29] Ferragina P, Vinciguerra G. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. Proc. of the VLDB Endowment, 2020, 13(8): 1162–1175. [doi: 10.14778/3389133.3389135]

[30] Galakatos A, Markovitch M, Binnig C, Fonseca R, Kraska T. FITing-tree: A data-aware index structure. In: Proc. of the 2019 Int'l Conf. on Management of Data. Amsterdam: ACM, 2019. 1189–1206. [doi: 10.1145/3299869.3319860]

[31] Chen JS, Chen K, Shou LD, Jiang DW, Chen G. ALERT: Workload-adaptive learned index based on radix tree. Ruan Jian Xue Bao/Journal of Software, 2022, 33(12): 4688–4703 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6354.htm [doi: 10.13328/j.cnki.jos.006354]

[32] Tang CZ, Wang YY, Dong ZY, Hu GS, Wang ZG, Wang MJ, Chen HB. XIndex: A scalable learned index for multicore data storage. In: Proc. of the 25th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. San Diego: ACM, 2020. 308–320. [doi: 10.1145/3332466.3374547]

[33] Li PF, Hua Y, Jia JN, Zuo PF. FINEdex: A fine-grained learned index scheme for scalable and concurrent memory systems. Proc. of the VLDB Endowment, 2021, 15(2): 321–334. [doi: 10.14778/3489496.3489512]

[34] Wu JC, Zhang Y, Chen SM, Wang J, Chen Y, Xing CX. Updatable learned index with precise positions. Proc. of the VLDB Endowment, 2021, 14(8): 1276–1288. [doi: 10.14778/3457390.3457393]

[35] Li PF, Lu H, Zhu R, Ding BL, Yang L, Pan G. DILI: A distribution-driven learned index (Extended version). arXiv:2304.08817, 2023.

[36] O'Neil P, Cheng E, Gawlick D, O'Neil E. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351–385. [doi: 10.1007/s002360050048]

[37] Yang J, Kim J, Hoseinzadeh M, Izraelevitz J, Swanson S. An empirical guide to the behavior and use of scalable persistent memory. In: Proc. of the 18th USENIX Conf. on File and Storage Technologies. Santa Clara: USENIX Association, 2020. 169–182.

[38] Liu XY, Lin ZJ, Wang HQ. Novel online methods for time series segmentation. IEEE Trans. on Knowledge and Data Engineering, 2008, 20(12): 1616–1626. [doi: 10.1109/TKDE.2008.29]

[39] Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual. https://www.intel.cn/content/www/cn/zh/developer/articles/technical/intel-sdm.html

[40] Intel Corporation. Deprecating the PCOMMIT Instruction. https://www.intel.cn/content/www/cn/zh/developer/articles/technical/deprecate-pcommit-instruction.html

[41] Wongkham C, Lu BT, Liu C, Zhong ZC, Lo E, Wang TZ. Are updatable learned indexes ready? Proc. of the VLDB Endowment, 2022, 15(11): 3004–3017. [doi: 10.14778/3551793.3551848]

附中文参考文献:

[22] 张洲, 金培权, 谢希科. 学习索引: 现状与研究展望. 软件学报, 2021, 32(4): 1129–1150. http://www.jos.org.cn/1000-9825/6168.htm [doi: 10.13328/j.cnki.jos.006168]

[25] 王中华, 赖必梁, 赵泽阳, 鲁凯, 万继光. APLI: 一种基于持久化内存的高性能学习索引. 小型微型计算机系统, 2024, 45(9): 2110–2118. [doi: 10.20009/j.cnki.21-1106/TP.2023-0140]

[31] 陈井爽, 陈珂, 寿黎但, 江大伟, 陈刚. ALERT: 基于 Radix Tree 的工作负载自适应学习型索引. 软件学报, 2022, 33(12): 4688–4703. http://www.jos.org.cn/1000-9825/6354.htm [doi: 10.13328/j.cnki.jos.006354]

张志国(1990－), 男, 硕士, 主要研究领域为数据库索引.

陈珂(1977－), 女, 博士, 副研究员, CCF 专业会员, 主要研究领域为非结构化数据管理, 数据挖掘, 隐私保护.

谢钟乐(1992－), 男, 博士, 研究员, CCF 专业会员, 主要研究领域为高性能索引, 联机分析处理, 新型数据库系统构建.

寿黎但(1974－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为数据库, 大数据智能.