软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007637] [CSTR: 32375.14.jos.007637]

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

HCA-Index: 基于数据冷热感知的学习型索引*

孙伊鸣, 信俊昌, 赵少杰, 操天童, 洪振强, 安 宸

(东北大学 计算机科学与工程学院, 辽宁 沈阳 110169)

通信作者: 信俊昌, E-mail: xinjunchang@mail.neu.edu.cn

摘 要: 大数据时代背景下, 传统索引 (如 B+树) 面临内存占用过高等问题, 学习型索引以其低内存占用和高查询效率正逐步替代传统索引. 然而, 现有学习型索引难以有效拟合多样化的数据分布, 且易受新数据插入导致分布变化的影响, 引发性能下降. 为解决这些问题, 提出一种基于数据冷热感知的学习型索引 HCA-Index. 该索引的核心包括: 设计基于误差阈值的渐进分区算法实现对数据分布的动态拟合; 通过自下而上提取键值范围并采用层级合并策略构建高精度索引; 设计动态演化的数据温度计算模型识别冷热数据; 利用节点级冷热分区支持数据迁移与快速查询. 在真实数据集上进行的实验表明, 相较于传统索引及最新的学习型索引, HCA-Index 在进一步降低内存占用的同时, 显著缩短查询延迟, 并有效减少新数据插入导致的重训练次数.

关键词: 学习型索引; 动态分区; 数据温度; 冷热分离

中图法分类号: TP18

中文引用格式: 孙伊鸣, 信俊昌, 赵少杰, 操天童, 洪振强, 安宸. HCA-Index: 基于数据冷热感知的学习型索引. 软件学报. http://www.jos.org.cn/1000-9825/7637.htm

英文引用格式: Sun YM, Xin JC, Zhao SJ, Cao TT, Hong ZQ, An C. HCA-Index: Learned Index Based on Data Hot-cold Awareness. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7637.htm

HCA-Index: Learned Index Based on Data Hot-cold Awareness

SUN Yi-Ming, XIN Jun-Chang, ZHAO Shao-Jie, CAO Tian-Tong, HONG Zhen-Qiang, AN Chen

(School of Computer Science and Engineering, Northeastern University, Shenyang 110169, China)

Abstract: In the era of big data, traditional indexes (e.g., B+ trees) face challenges, including high memory consumption. Learned indexes are increasingly replacing traditional indexes because of their lower memory consumption and higher query efficiency. However, existing learned indexes struggle to effectively adapt to diverse data distributions and are prone to performance degradation when distribution shifts occur due to new data insertions. To address these issues, this study proposes HCA-Index, a learned index based on data hot-cold awareness. The core components of HCA-Index include 1) designing a progressive partitioning algorithm based on error thresholds to dynamically fit data distributions; 2) constructing a high-precision index by extracting key ranges in a bottom-up manner and adopting hierarchical merging strategies; 3) designing a dynamically evolving data temperature calculation model to identify hot and cold data; and 4) leveraging node-level hot-cold partitioning to support data migration and fast querying. Experiments on real-world datasets demonstrate that, compared with traditional indexes and state-of-the-art learned indexes, HCA-Index significantly reduces query latency, further lowers memory consumption, and effectively reduces the number of retraining operations caused by new data insertions.

Key words: learned index; dynamic partitioning; data temperature; hot-cold separation

大数据技术日益普及, 无论是对于传统的关系型数据库 (如 MySQL 等) 还是对于 Hadoop 分布式文件系统 (Hadoop distributed file system, HDFS), 如何高效管理频繁更新的数据面临巨大挑战. 例如, 在面向算力的分布式联邦湖仓系统中, 分散的数据湖之间通过联邦层实现数据共享. 然而, 各个数据湖不仅要管理和共享各类智能终端产生的表格、文本、图片、视频、语音等结构化、半结构化、非结构化数据, 还要共享联邦湖仓系统的网络状态数

* 基金项目: 国家自然科学基金 (62432003); 国家大学生创新创业训练计划 (251347)
收稿时间: 2025-06-14; 修改时间: 2025-10-20; 采用时间: 2026-01-19; jos 在线出版时间: 2026-06-03

据、资源监控数据以及系统日志数据等信息. 对联邦湖仓数据的查询和更新操作不仅频繁、复杂, 且存在明显的访问时间规律以及访问热点规律. 因此, 结合数据的查询和更新频率以及数据访问热点规律对数据进行冷热数据感知并分离对于数据库的存储具有重要意义.

索引通过为每条记录的键值建立额外的映射关系, 使得系统无需全表扫描即可快速定位目标数据[1,2]. 但索引同时需要额外内存空间用于存储数据库表中键 (key) 的排序结果, 以及对应数据在物理标识数据页中的逻辑指针列表. 传统索引技术面临爆炸式增长的数据, 索引的内存开销随之增加, 查询速度也变得越来越慢, 并且传统索引面向的是泛型数据, 没有利用底层数据的分布规律, 特定情况下的效率有待优化. 为了解决这些问题, 学习型索引的概念被提出, 并成为当今数据库领域的热点[3,4].

学习型索引使用机器学习模型取代传统的索引结构, 通过学习数据的分布规律, 建立键与其存储位置之间的映射关系, 从而提高查询效率并降低内存占用. 例如, 最早期的学习型索引 (recursive model index, RMI)[5]采用多层模型结构, 通过上层模型选择下层模型, 逐层递归, 最终预测键的位置. 然而, 早期的学习型索引主要针对静态数据, 对于频繁更新的数据场景, 模型需要频繁重新训练, 导致性能下降. 进一步地, 一些方法试图通过缓存策略或数据分组优化, 如 Doraemon[6]提出热键预测机制, CARMI[7]引入基于成本的构建算法和缓存感知设计, 以优化索引结构; 其他如 LIFOSS[8]、FLIRT[9]则针对流数据和窗口索引场景设计轻量策略, 规避全局重建. 然而, 这些方案往往侧重于局部优化, 尚未根本性解决频繁数据更新带来的性能退化问题.

针对这一问题, 研究者提出了多种改进方案以应对数据更新. 例如, ALEX[10]通过在每个叶子节点预留间隙, 减少插入操作时的数据移动; PGM-index[11]为每个段添加缓冲区, 处理更新操作; LIPP[12]通过向下分裂创建新节点, 处理插入导致的冲突. 然而, 这些方法在处理数据更新时, 往往会改变当前数据的分布, 为了使模型保持准确, 通常需要频繁改变索引结构, 甚至定期在线重新训练, 从而导致性能受限. 具体而言, ALEX 节点触发阈值后会对此节点进行分裂并分别进行重新训练, 其分裂不仅会进一步降低索引整体查询性能, 而且由于分裂后数据分布改变, 对其进行重新训练也是比较耗时的. LIPP 采用向下分裂创建新节点会使得查询路径增长, 进而造成查询效率降低. PGM-index 通过维护多个索引, 在插入更新时需要合并之前所有索引并重建, 重建索引不仅代价高, 也会造成查询性能下降. 由于模型重训练的代价很高. 并且随着新数据涌入而改变索引结构也会使总体查询效率下降. 因此, 需要研究新的方法进一步减少模型重训练代价的同时, 尽量维持索引结构的高性能. 此外, 上述方法几乎没有考虑数据冷热 (访问频率) 的问题. 因此, 本文提出一种基于数据冷热感知的学习型索引 (learned index based on data hot-cold awareness, HCA-Index), 从数据冷热特征出发, 尝试解决上述问题, 通过将冷数据迁移至冷区而热区数据位置保持不变的策略, 避免频繁改变索引结构, 由于热区数据仅占原本数据的一小部分, 因此重新训练代价大大降低. 在保证热区进行有效查询的同时, 减小了模型重训练与索引结构频繁改变的代价. 本文主要贡献总结如下.

(1) 提出基于误差驱动的自适应渐进式分区算法. 与传统的静态分区策略不同, 该算法通过动态回归模型调整机制, 能够实时感知数据分布规律的变化趋势, 结合增量学习优化方法, 将算法时间复杂度从 $O(n^2)$ 降低至 $O(n)$. 实验表明, 在相同时间预算下, 新算法实现分区数量显著减少, 单个分区数据容量显著提升, 线性回归拟合的预测误差低于传统方法, 显著增强了对数据分布变化的鲁棒性.

(2) 设计基于范围合并的层级式索引结构. 区别于传统方法直接基于原始数据构建索引, 本方法采用自底向上的键值范围提取策略, 通过建立分区边界特征向量, 构建具有空间拓扑特征的层级索引. 由于键值范围数量较原始数据缩减数个数量级, 索引训练数据量显著下降, 上层索引构建速度相应提升. 同时, 由此方法构建的索引模型仅关注数据的键值范围分布, 使得索引结构对数据动态变化的适应能力大大提升.

(3) 建立动态温度感知的冷热数据管理模型. 引入基于滑动窗口的访问频率量化方法, 构建具有时变特性的温度演化方程, 实现冷热数据的精准识别与动态迁移. 热数据分区采用高精度线性回归建模, 冷数据存储于低维护成本的独立分区, 在保证总体查询性能的前提下, 对热区数据拟合误差远远低于对全部数据的拟合误差, 查询响应速度显著提升. 结合范围合并策略, 新结构仅需更新底层的模型参数即可适应数据分布变化, 显著减少了模型调整开销, 有效避免高频数据更新引发的内存开销大和查找速度慢问题.

(4) 在真实数据集 SOSD (https://github.com/learnedsystems/SOSD) 上的大量实验表明, 相比于传统算法

(B+树) 和前人工作提出的若干学习型索引, HCA-Index 的查询性能显著提升, 优于传统索引及现存的学习型索引. 同时, 其索引空间占用相比现有学习型索引实现了数量级优化, 构建速度也有一定的优势. HCA-Index 对不同分布的数据展现出良好的适应性, 不仅大幅提高了拟合模型的重训练效率, 而且有效减少了因大规模新数据插入导致的重训练次数.

1 相关工作

1.1 学习型索引的提出

B+树是数据库系统中常见的一维索引结构, 根据机器学习的思想, 可以把 B+树看成一个模型, 输入键值, 输出对应此键值数据的位置. 由于 B+树常索引一个内存块或磁盘页, 而不是单个数据, 这一特性让 B+树可以看作一个回归树, 输出一个具有最大误差范围的数组位置. 在此基础上, Kraska 等人[5]提出了 RMI 学习型索引, 采用一组模型有效的近似累积分布函数, 使用此模型预测给定键值在数组中的位置. 由于 RMI 利用了数据的分布特点, 不会涉及 B+树查找数据时频繁的比较操作, 以及 RMI 仅存储模型的斜率和截距, 在查询性能和存储性能方面显著优于传统索引. 尽管 RMI 索引在查询性能和存储性能方面相比于传统的学习型索引优势显著, 但仍存在一些问题. 首先, 构建 RMI 索引需要训练机器学习模型, 这个训练过程相比于传统的索引构建时间有所增加. 同时, RMI 索引在设计索引结构时并没有考虑插入新数据的情况, 因此其并不支持新数据的插入[13,14]. 针对上述问题, 已经有研究对 RMI 学习型索引进行优化和改进.

1.2 现有的改进与优化策略

(1) 插值法 (RadixSpline): 相比 RMI 结构的学习型索引, 基于分段线性函数的索引通过一趟数据扫描即可完成构建, 并支持自下而上的构建方式. 插值法同样具备这些优势. RadixSpline[15]使用样条插值拟合数据分布规律, 并借助基数树来索引这些样条点. Setiawan 等人[16]进一步探索了切比雪夫插值法和伯恩斯坦插值法在学习型索引中的应用. 此方法只优化了 RMI 索引的构建过程, 其仍没有解决索引不支持插入的问题.

(2) 基于缓冲区的异地插入策略 (FITing-Tree): FITing-Tree[17] 采用基于贪心算法的分段方法, 将数据划分为由分段线性函数拟合的若干段. 分段过程会预先设定一个误差阈值, 确保段内数据的预测位置都能落在其真实位置的误差范围内. 每个分段均配置固定大小的缓冲区. 插入数据时, 首先写入对应分段的缓冲区; 当缓冲区填满时, 将该缓冲区与其关联的数据段合并, 并重新执行分段算法. 模型的顶层结构借鉴 B+树的构建方法, 重新分段并更新上层索引的过程与 B+树类似. 这种基于缓冲区的方法虽然解决了索引无法插入新数据的问题, 但由于查询时需先查找缓冲区, 降低了整体的查询效率, 同时也带来缓冲区合并重训练等一系列问题.

(3) 就地插入策略 (ALEX): ALEX 提出在叶子节点的排序数组中预留空隙 (gap). 当需要插入新数据时, 若预测位置准确, 则直接插入该位置的空隙; 若预测不准确, 则通过指数搜索找到正确的插入位置. 若目标位置为空隙, 则直接插入; 若非空, 则移动数据以腾出空隙后再插入. 此外, ALEX 的 RMI 模型支持动态更新: 当叶子节点因数据量超出阈值而分裂时, 它转变为内部节点, 新分裂产生的若干间隙数组则作为叶子节点挂载在其下. 就地插入策略相较设置缓冲区的方法插入效率更高, 也避免了查询时需要先查找缓冲区的问题, 本索引在插入时也采取就地插入策略, 并结合冷热数据迁移来实现索引的动态更新.

(4) 基于 LSM 树的插入策略与模型压缩 (PGM-index): PGM-index[11]借鉴 LSM 树[18]的思想分层管理数据, 在每一层托管一个学习型索引模型, 仅在触发合并操作时才更新相应模型. PGM-index 还提出了模型的压缩方案: 对分段起始键进行无损压缩 (可利用成熟的排序数据子集压缩技术), 并对斜率应用专门设计的无损压缩算法. 基于 LSM 树的插入策略导致存在多个学习型索引, 查询效率较低, 同时也加大了内存占用, 但是 PGM-index 利用无损压缩算法缓解了这一问题.

(5) 并发数据结构 (XIndex): XIndex[19]的数据结构设计支持高并发操作. 它整合了现有的乐观并发控制、细粒度锁和 RCU (read-copy-update) 技术, 并设计了一种两阶段合并算法来支持并发. 在合并阶段, 会建立新组、冻结旧缓冲区, 并将新数据暂存于临时缓冲区. 在随后的复制阶段, XIndex 将新排序数组中的每个记录指针原子性地

替换为最新值, 最终回收旧组所占用的内存资源. 并发数据结构在高并发环境下性能良好, 但可更新学习型索引中只有少数索引支持高并发场景, 本文也在单线程环境下进行实验, 因此不对其进行具体实验.

(6) 精准预测型索引 (LIPP、DILI[20]): 以往的学习型索引难以实现完全精准的预测, 因此在定位实际数据时仍需结合二分查找 (如 FITing-Tree 所采用) 或指数搜索 (如 ALEX 所采用) 等辅助搜索策略. 针对这一问题, LIPP 索引提出了一种基于多级节点的精准预测机制: 当预测位置出现冲突时, 不直接对键进行移位调整, 而是将该位置转换为指向下级节点的指针, 并通过创建新的节点层级来化解冲突. 尽管该方法实现了精准预测, 但较长的冲突链会导致内存占用显著增加. 为缓解这一不足, DILI 引入了叶子节点本地优化策略, 仅在叶子节点层面采用类似 LIPP 的多级节点构建方法以实现精准预测. 同时, DILI 采用两阶段构建流程: 首先通过自底向上的方式构建 BU-tree 以确立索引的整体骨架; 随后自上而下地对节点进行划分, 强制使子节点均匀分割键值区间. 得益于内部节点键值范围的均匀分布特性, DILI 实现了内部节点层面的精准预测.

(7) 学习式划分数据分区 (Dabble[21]): 直接使用单一模型学习全局数据分布存在一定困难. 为此, Dabble 首先利用 K-means 聚类算法对数据进行初步划分, 然后在各个数据子集上分别训练神经网络, 并在损失函数中引入数据访问热度, 使得神经网络对高频访问数据具备更优的查询性能. 此外, Dabble 通过模型解耦设计, 在向某一区域插入新数据时, 仅影响该区域的数据分布, 无需重新训练其他区域的模型. 然而, Dabble 虽考虑了数据访问热度, 却难以动态捕捉冷热数据的变化规律. 旧的热点数据尽管历史访问频次高, 但随着时间推移, 访问热点可能已转移至其他数据, 而 Dabble 中的神经网络无法自动识别此类变化. 针对这一局限, 本文所提出的索引结构设计了动态温度识别机制, 能够自适应地响应数据访问热点的变化. 同时, 在模型重新训练过程中, 本文方法同样保证不影响其他分区模型, 实现了有效的模型解耦.

表 1 对比了传统索引 (B+树)、只读学习型索引 (RMI、RadixSpline) 以及其他可更新学习型索引在数据划分、搜索策略、插入策略 (只读学习型索引不讨论) 及其他优化措施方面的异同. 标注星号 (*) 的索引将在下文实验中作为基准索引组进行对比. 尽管上述方法对传统的 RMI 索引进行了改进, 学习型索引仍面临一定的困难和挑战. 首先, 现有很多方法对数据分区进行静态划分, 这种划分方式限制了索引结构的泛化能力. 此外, 对数据库的查询和更新操作存在明显的访问时间规律以及访问热点规律, 现有学习型索引技术在更新过程中大多没有考虑数据的访问频率 (冷热), 制约学习型索引的性能进一步提升. 因此, 本索引将数据冷热与现有的插入方法相结合, 以实现对热区数据更高效的更新方法.

表 1　现有一维索引对比

名称	数据划分	搜索策略	插入策略	其他
*B+ tree[1]	不划分	自上而下遍历	就地插入	—
*RMI[5]	不划分	线性回归预测	不支持	—
*RadixSpline[15]	不划分	样条插值+基数树	不支持	一趟构建索引
FITing-Tree[17]	贪心算法	分段线性回归预测	异地插入 (缓冲区)	固定误差
*ALEX[10]	等分	线性回归预测	就地插入 (空隙数组)	空隙数组+指数搜索
*PGM-index[11]	贪心算法	分段线性回归预测	异地插入 (LSM树)	模型压缩
XIndex[19]	分组 (等分)	线性回归预测	异地插入 (缓冲区)	高并发+两阶段合并
*LIPP[12]	不划分	核化模型预测	就地插入	多级节点精准预测
DILI[20]	BU-Tree+平均分割	线性回归预测	异地插入 (LSM树)	叶子节点本地优化
Dabble[21]	K-means	神经网络预测	异地插入 (LSM树)	结合查询频率训练
*SWIX[22]	时间窗口分段	线性回归预测	就地插入 (滑动窗口)	无锁并发设计

2 HCA-Index 的构建与更新

针对上述问题, 提出一种基于数据冷热感知的学习型索引 (hot-cold-awareness learned index, HCA-Index). HCA-Index 不仅实现了动态的数据分区, 而且在每个分区中依据数据温度, 对冷热数据分离存储和管理. HCA-

Index 的工作流程如图 1 所示, 对于一个新的待构建索引数据, 首先通过基于误差阈值的渐近分区算法确定该数据应该插入至哪个数据分区 (该分区对应学习型索引的叶子节点), 随后, 采用自底向上的方式更新当前的索引结构. 其次, 在该数据分区内, 确定数据插入的准确位置, 在经过校验后插入该数据, 新插入的数据一定存储在分区的热区中. 数据分区中的每一个数据有一个数据温度属性, 当数据分区中的热区达到上水位阈值时, 将一部分热区中温度较低的数据迁移至改数据分区的冷区. 当查询一个键时, 首先依据当前的索引结构确定要查找键的数据分区, 随后先遍历改数据分区的热区查找, 如果没有找到则遍历冷区, 最后返回键的位置或查找失败.

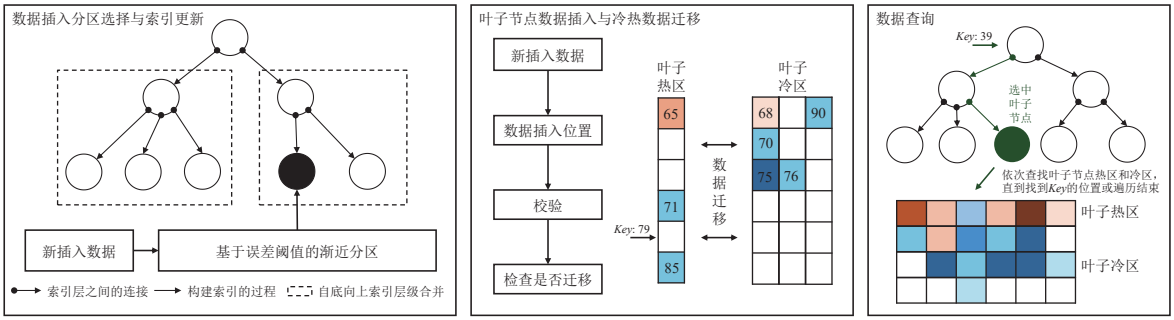


图 1 HCA-Index 概况

2.1 索引的构建过程

遵循数据库范式理论的索引架构设计是平衡查询吞吐量与插入延迟优化的核心要素. 针对现有索引在动态数据分布场景中适应性不足的痛点, 本研究提出基于自底向上 (bottom-up) 构建范式的误差驱动型渐进分区算法, 该算法通过渐进式分区扩展机制与动态范围合并策略, 构造具有自适应调整能力的多级树状索引结构.

2.1.1 基于误差阈值的渐近分区策略

本文提出一种基于误差驱动的渐进式分区优化算法 (progressive partitioning algorithm, PPA), 其核心设计需满足 3 个关键标准: 在确保模型拟合误差可控的前提下, 实现最少的分区数量、最大化分区内数据容量以及最小化分区空置规模. PPA 算法通过融合贪心策略与增量学习方法实现上述目标. 首先, 设定预置的误差容忍阈值如公式 (1) 所示, 此阈值用于判断后续数据点能否纳入当前分区.

$$\left| x_m - \frac{y_m - intercept_{m-1}}{slope_{m-1}} \right| \leqslant \epsilon \tag{1}$$

其中, x_m 和 y_m 为第 m 个被纳入当前分区的数据点坐标, $slope_m$ 和 $intercept_m$ 为纳入第 m 个数据点后当前分区的线性模型参数 (斜率和截距), ϵ 为人为设定的误差界限. 初始化阶段采用基础线性模型对起始数据集使用公式 (2) 进行拟合.

$$slope_0 = \frac{y_1 - y_0}{x_1 - x_0},\ intercept_0 = y_0 - slope_0 \times x_0 \tag{2}$$

随后按照数据时序逐步纳入符合当前模型误差约束的后续数据点. 每当新数据点被合并入当前分区后, 系统立即启动增量学习机制由先前线性方程的统计量 $\bar{x}_{m-1}$、$\bar{y}_{m-1}$、S_{xy}^{m-1}、S_{xx}^{m-1} 递归计算得到新线性方程统计量 $\bar{x}_m$、$\bar{y}_m$、S_{xy}^{m}、S_{xx}^{m}, 如公式 (3)–(6) 所示:

$$\bar{x}_m = \frac{(m-1)\bar{x}_{m-1} + x_m}{m} \tag{3}$$

$$\bar{y}_m = \frac{(m-1)\bar{y}_{m-1} + y_m}{m} \tag{4}$$

$$S_{xy}^{m} = S_{xy}^{m-1} + (x_m - \bar{x}_{m-1})(y_m - \bar{y}_{m-1})\frac{m-1}{m} \tag{5}$$

$$S_{xx}^{m} = S_{xx}^{m-1} + (x_m - \bar{x}_{m-1})^2\frac{m-1}{m} \tag{6}$$

其中, $\bar{x}_m$ 和 $\bar{y}_m$ 为纳入第 m 个点后的数据点均值. S_{xy}^m 和 S_{xx}^m 为纳入 m 个点后的数据点协方差项和方差项. 最后根据公式 (7) 由更新后的统计量更新线性方程参数, 这种递推式参数更新可使模型渐进适应新增数据分布特征. 随着模型精度的迭代提升, 其继续识别和吸收符合更新后误差约束的附加数据点, 从而形成自我强化的数据吸纳机制. 当经过充分迭代后无法继续吸收新数据点时, 当前分区持续过程终止, 算法将初始化新分区迭代流程.

$$slope_m = \frac{S_{xy}^m}{S_{xx}^m},\ intercept_m = \bar{y}_m - slope_m \cdot \bar{x}_m \tag{7}$$

相较于传统批量建模方法, 本算法通过增量学习实现了核心过程的在线更新, 将模型参数优化计算复杂度从 O(n) 降低至 O(1). 在保证各分区数据规模最大化的同时, 该设计不仅在总体上压缩了分区数量, 而且通过参数渐进优化动态拟合数据分布, 显著降低了模型的驻留误差.

例 1: 如图 2 所示, 基于两个初始数据点构建初始线性方程, 通过误差范围 (error bound) 评估确定有效区域, 初始模型成功覆盖 D1–D3 这 3 个邻近数据点, 将 D1–D3 纳入分区并增量训练原线性方程, 优化后的新模型在参数空间重新定位, 最终成功将先前不符合条件的数据点纳入范围.

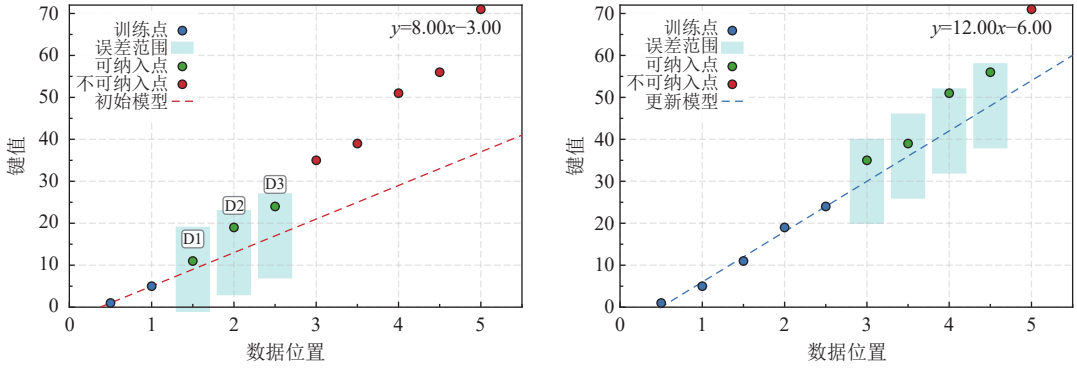


图 2 渐进分区算法演示

2.1.2 基于范围合并构建索引

HCA-Index 的完整构建过程如后文图 3 所示. 在实现基于误差阈值的数据分区之后, 可确定待插入键应该插入到哪个数据分区, 即学习型索引的哪个叶子节点. 下面讨论如何基于范围合并的方式自底向上构建索引结构. 在基于误差阈值的数据分区之后, 得到若干个范围互不重叠的底层分区, 为了上层模型在得到一个键值时能够精准预测到下层分区的序列号, 应为每个底层分区提取出一个代表值, 最值在底层分区的一端, 若用最值训练上层模型无法有效预测分区每个数据, 因此每个底层分区存储时同步记录键值中位数, 形成元数据基础单元. 将底层分区的元数据特征作为新输入, 递归调用渐进分区算法实施二次分区操作. 该过程通过合并相邻分区的键值区间, 生成覆盖范围更大的上层分区, 同时训练对应的分区预测模型. 新生成的分区集合按照层级关系存储在索引结构的次高层, 形成树型索引的中间节点层. 此分层聚合过程通过迭代机制持续进行, 直至达到预设的最终层级深度. 最后进行顶层模型拟合, 最终层级的超大规模分区已实现对原始数据的全部覆盖, 此时直接采用线性回归模型对全局键值分布进行建模. 该模型作为索引结构的根节点, 在低内存占用的同时具备对查询请求的快速响应能力. 基于键值范围提取与层级合并的训练策略相较于传统直接从原始数据训练模型的方法具有显著优势. 具体而言, 其通过有效缩短误差传播路径, 减小了预测误差. 同时其大大减少了上层模型训练数据量, 从而显著提高了索引构建速度并减少了上层模型的内存开销.

2.1.3 误差传播路径验证

设数据键值 y 与其存储位置 x 的物理映射满足线性关系如公式 (8) 所示, 传统方法 (如 RMI 索引) 通过线性回归模型预测位置估计值 $\wedge x$, 再根据 $\wedge x$ 值确定数据所属分区. 当参数估计误差满足 $\Delta\beta_1 \ll \beta_1$ 时位置预测误差方差

可近似为公式 (9). 该方法存在两阶段误差传播: 1) 回归模型参数估计误差 β_0, $\Delta\beta_1$ 导致的 $\wedge x$ 偏差; 2) 通过 $\wedge x/width$ 计算分区号时的离散化误差.

$$y=\beta_0+\beta_1 x+\epsilon,\ \epsilon\sim\mathcal{N}(0,\sigma^2) \tag{8}$$

$$Var(\Delta x)\approx\frac{\sigma^2+Var(\Delta\beta_0)+x^2Var(\Delta\beta_1)}{\beta_1^2} \tag{9}$$

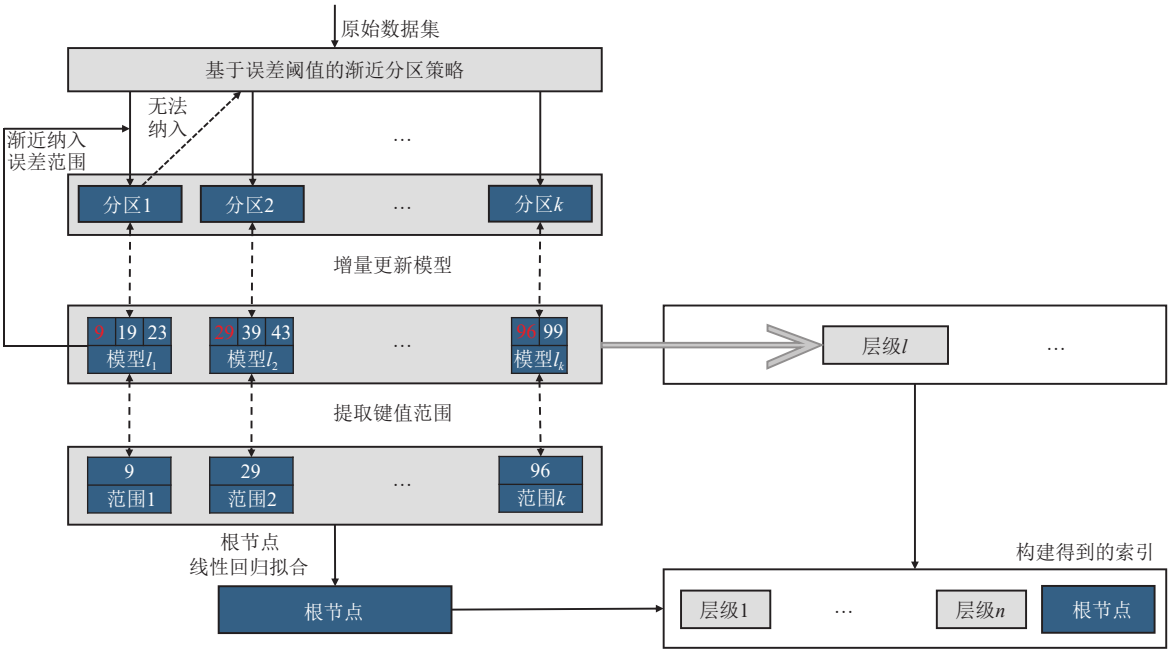


图 3 HCA-Index 的构建过程

由公式 (10) 定义分区序号 i 对应的最小键值边界, 其中 a 为基准键值偏移量, b 为分区间隔系数. 通过预训练获取 (a,b) 参数后, 可直接建立键值 y 到分区号 $\wedge i$ 的映射关系, 当分区间隔系数估计误差满足 $\Delta b\ll b$ 时, 分区预测误差方差可简化为公式 (11). 该方法直接预测分区号, 缩减了误差传播过程.

$$y_{\min}(i)=a+bi \tag{10}$$

$$Var(\Delta i)\approx\frac{\sigma^2+Var(\Delta a)+i^2Var(\Delta b)}{b^2} \tag{11}$$

传统方法的误差放大因子为 β^{-2}, 对斜率 β_1 高度敏感, 若 y 随 x 增长缓慢 ($\beta_1\to 0$), 误差急剧放大, 而键值范围提取方法的误差放大因子为 b^{-2}, 对分区间隔 b 敏感, 误差与 b^{-1} 成正比, 分区间隔 b 越小, 误差越大. 而本文先前提出的误差渐进分区算法保证了底层分区间隔尽量大, 同时层级合并的构建策略将上层分区间隔不断合并扩大, 大大地降低了模型的误差.

2.1.4 基于冷热信息的重构建

在某些应用场景中, 冷热数据的分布相对稳定, 在较长一段时间内基本不发生变化. 例如, 在时间序列数据和历史归档场景中, 近 3 个月的金融交易记录与 3 年前的记录, 其冷热属性往往具有明显的稳定性; 又如地理空间数据场景, 在提供地图服务时, 热门城市区域与冷门区域的访问热度也呈现稳定的冷热特征; 再如商品数据场景中, 畅销商品与冷门商品的冷热属性同样较为固定. 基于这一特点, 可利用历史冷热信息对索引进行离线重构. 在重构过程中进行叶子层分区时, 每个数据结构体 Data 中均记录有反映其冷热属性的温度值 T. 在执行渐进式分区算法时, 将温度 T 作为权值引入, 使得训练得到的线性回归模型能够对相对热点的数据具有更好的预测性能. 由于在分区过程中加入了温度权值, 最终得到的数据分区也更符合实际的冷热分布特征. 叶子层分区完成后, 可计算各叶子

节点的平均温度, 即节点内所有数据的温度之和除以节点所包含的数据总量. 将该平均温度作为叶子节点的权值, 并递归执行上述分区与赋权过程, 最终可构建出符合冷热分布特性的完整索引结构.

2.2 索引的更新策略

对于支持数据插入的学习型索引来说, 随着插入数据规模的持续扩张, 其查询性能呈现衰减趋势, 同时内存占用率与预测误差亦呈显著上升态势. HCA-Index 采用冷热数据识别与差异化存储策略, 以适度降低冷数据查询效率为代价, 优先保障热数据查询性能与预测误差的精准优化, 并实现整体内存开销的减少. 索引的更新流程如图 4 所示, 首先由 HCA-Index 索引得到插入位置, 检测是否超出阈值, 并对数据进行温度计算 (第 2.2.1 节) 后冷热数据划分与迁移 (第 2.2.2 节).

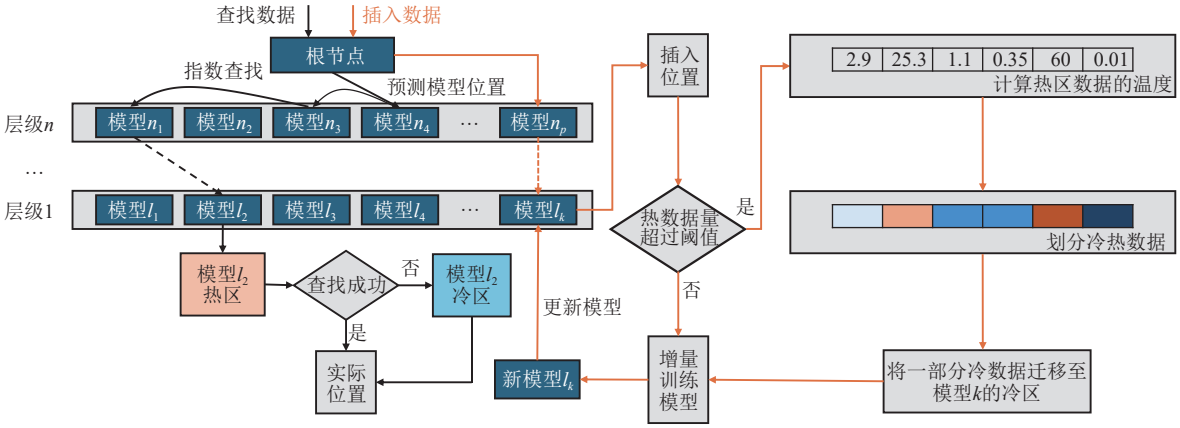


图 4 HCA-Index 数据查找与插入流程

2.2.1 冷热识别方法

在数据密集型应用场景中, 冷热数据的有效识别是提升存储系统性能的关键技术. 现有方法主要分为 3 类: 基于时间局部性的替换策略 (如 LRU、Clock 算法)、基于频率统计的淘汰机制 (如 LFU 算法) 以及基于机器学习的预测模型 (如 Transformer、LSTM). 然而, 这些传统方法在适应学习型索引 (learned index) 架构方面存在显著局限性: 前两类方法的数据频繁迁移特性会导致回归模型的拟合失效, 而深度学习模型的高训练开销难以满足实时性需求. 本研究针对学习型索引架构的特性需求, 提出基于动态温度演化的冷热识别模型 (dynamic thermal recognition model, DTRM). 本模型在牛顿冷却定律框架基础上, 通过重构衰减函数和设计基于历史访问次数的加热函数实现数据的冷热识别.

基于牛顿冷却定律的数据温度计算方法如公式 (12) 所示, 其中 $T(t_n)$ 表示在 t_n 时刻此数据的温度, α 为温度衰减系数, $heat$ 为加热系数.

$$T(t_n) = T(t_{n-1})\mathrm{e}^{-\alpha(t_n - t_{n-1})} + heat \tag{12}$$

该模型采用指数衰减机制保证热数据的时效性, 但存在两个缺陷: 指数函数的衰减特性表现为初期剧烈下降、后期平缓衰减, 这与数据价值衰减的客观规律相悖. 在实际场景中, 新近访问数据在短期内应保持较高温度值以反映其潜在访问价值, 只有当持续未访问时才应加速衰减; 固定值 $heat$ 的加热项忽视了数据的历史访问价值差异. 实际系统中, 频繁访问的热数据应获得更强的加热激励, 而偶发访问的冷数据则需要限制其温度增益.

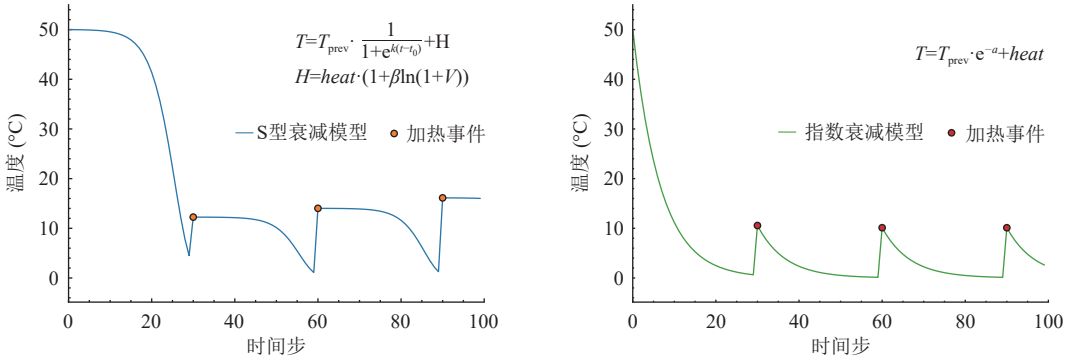
针对以上问题, HCA-Index 采用一种新的数据温度计算方法如公式 (13) 和 (14) 所示:

$$T_S(t_n) = \begin{cases} T_S(t_{n-1}) \cdot \dfrac{1}{1+\mathrm{e}^{k\left(t_n - t_0^{(n-1)}\right)}} + heat \cdot (1+\beta \cdot \ln(1+V(t_{n-1}))), & \text{if 访问} \\ T_S(t_{n-1}) \cdot \dfrac{1}{1+\mathrm{e}^{k\left(t_n - t_0^{(n-1)}\right)}}, & \text{otherwise} \end{cases} \tag{13}$$

$$t_0^{(n)} = \begin{cases} t_n + t_0^{\text{init}}, & \text{if 访问} \\ t_0^{(n-1)}, & \text{otherwise} \end{cases} \tag{14}$$

其中, $T_S(t_n)$ 表示在 t_n 时刻此数据的温度, k 为温度衰减系数, β 为数据加热系数, $heat$ 为数据加热基础值, $V(t_{n-1})$ 表示截至 t_{n-1} 时刻此数据被访问的次数, t_0^{init} 为初始衰减中点, $t_0^{(n)}$ 为动态调整的第 n 步衰减中点. 在每个数据结构体 Data 中, 内置了一个时间戳用于记录上次访问时间, 以及一个计数位用于记录被访问次数. 当数据被访问时, 系统首先根据公式 (13) 中的第 1 条规则更新当前温度, 并将其记录到结构体 Data 的温度项中, 同时依据公式 (14) 更新衰落中点. 直至执行数据迁移时, 系统将对所有热区数据调用公式 (13) 中的第 2 条规则重新计算其温度, 并依据结果进行迁移. 具体的迁移机制详见第 2.2.2 节.

例 2: 如图 5 所示, S 型衰减曲线在访问后的 $t_n < t_0^{(n)}$ 区间保持平缓衰减, 当 $t_n \geqslant t_0^{(n)}$ 时触发指数级衰减. 这种双阶段特性既保证了潜在热数据的温度维持能力, 又实现了长期未访问数据的快速降温. 通过公式 (14) 构建 $t_0^{(n)}$ 与历史访问次数 $V(t_{n-1})$ 的正相关关系, 使得高价值数据的衰减中点随访问频次动态后移, 形成自适应的温度维持窗口. 加热项 $heat \cdot (1+\beta \cdot \ln(1+V(t_{n-1})))$ 通过 $V(t_{n-1})$ 编码历史访问价值, 确保频繁访问数据获得高温度增益; 同时, 采用对数函数有效抑制高频访问数据的温度累积, 避免数据温度过高.



(a) S型衰减模型温度动态 (b) 指数衰减模型温度动态

图 5 动态温度识别模型演示

2.2.2 插入过程

HCA-Index 是一种支持新数据插入的索引. 在每次插入数据之后只需要更新一部分索引参数即可而不需要重训练整个索引模型. 数据的插入过程如算法 1 所示, 首先根据点查询过程搜索到键值所属叶子层分区 (第 1–7 行), 在叶子节点内部, 线性回归模型 M_{leaf} 将输出待插入键的理论存储位置 p_{pred} (第 8 行). 若该位置存在空闲间隙, 则向相邻位置执行指数步长跳跃式探测直到找到首个非空位置, 在最终的间隙插入操作前, 执行校验, 使左侧相邻键值小于待插入键值, 右侧相邻键值大于待插入键值, 通过不等式约束 $k_{\text{prev}} < k_{\text{new}} < k_{\text{next}}$ 维持存储结构全局单调递增特性, 确保索引逻辑一致性 (第 10 行).

算法 1. 插入过程 *insert*(*k*, *layers*).

输入: 插入键 *k*, 索引层结构 *layers*;
输出: 插入是否成功.

1. *model* ← *root*
2. for $l = L-1 \rightarrow 0$ do /* 从上层 $(L-1)$ 向下层遍历, 直到叶子层 (第 0 层) */
3. predict *idx* ← (*k* − *model.b*)/*model.k* /* 对当前键 *k* 预测, 得到下一层的索引位置 *idx* */
4. *idx* ← *clip*(*idx*, 0, |*layer.models*| − 1)

5. *model* ← *exponentialSearch*(*layer.models*, *k*, *idx*) /* 用指数搜索快速查找包含 *k* 的子模型 */
6. if *model* == null then return null
7. if *l* == 0 then /* 到叶子层 */
8. predict *pos* ← (*k* − *model.b*)/*model.k*
9. *pos* ← *clip*(*pos*, 0, |*model.data_array*| − 1)
10. *ret* ← *leaf_insert*(*model*, *k*, *pos*) /* 调用叶子插入函数尝试插入数据 */
11. return &*model.data_array*[*ret*] /* 返回插入后键所在的地址 */
12. return null

为了保证叶子分区的冷热数据量适中, HCA-Index 采用间隙数组作为叶子节点的底层存储结构, 其冷热数据迁移机制通过动态温度演化的冷热识别模型与水位检测相结合实现. 为精确控制数据迁移触发时机以及维持合理的热区数据数量, 设计双阈值检测机制, 高水位阈值用于判断何时进行数据迁移, 低水位阈值用于保证热区数据数量, 其中高水位阈值 (θ_{high}) 设定为数组总容量 C_{total} 的 80%, 低水位阈值 θ_{low} 设置为 $0.2\times N_{\text{active}}$ 防止热区数据数量过少. 当数据迁移后需满足 $N_{\text{active}} \leqslant \theta_{\text{low}}$, 确保系统进入稳定态. 当触发高水位条件时, 遍历间隙数组所有有效数据项 $a_1, a_2, \ldots, a_{N_{\text{active}}}$, 调用动态温度演化的冷热识别公式计算各元素的温度值, 生成温度序列 $T_1, T_2, \ldots, T_{N_{\text{active}}}$, 随机化快速选择算法可以在无序数组中以 O($n$) 的时间复杂度选出第 K 大的数据, 并通过随机选择枢轴避免最坏的 O(n^2) 情况, 温度序列实际上是一个无序数组, 因此应用随机化快速选择算法在温度序列中定位第 $0.2\times N_{\text{active}}$ 大的温度值作为分界阈值 T_{thres}, 根据算法 2 建立冷数据暂存区 C_{temp} (第 1 行), 遍历原数组执行温度比对, 若 $T_i < T_{\text{thres}}$, 将数据项标记为冷数据, 移入 C_{temp} 并释放原存储位 (第 6、7 行), 若 $T_i > T_{\text{thres}}$, 保留为热数据并维持原位存储 (第 9 行). 最后进行冷区合并及热区再训练过程, 系统将待迁移的冷数据集合 C_{new} (已按升序排列) 与历史冷区数据集 C_{old} (若存在) 通过双指针归并算法合并为全局有序的新冷区 C_{merged} (第 18、19 行), 同时针对保留在热区的数据子集, 通过增量学习得到全新拟合参数, 更新原有的线性回归模型 (第 10–17 行).

算法 2. 迁移过程 *process_leaf_model*(*leaf*, *cold_vector*, *threshold_t*).

输入: 待插入叶子节点 *leaf*, 对应冷区 *cold_vector*, 温度阈值 *threshold_t*;

1. *temp_container* ← new vector() // 存储过时数据
2. *training_points* ← new vector() // 存储有效数据
3. for each *data* in *leaf.data_array*: // 分离过时数据与有效数据
4. if *data.y* 无效 continue
5. if *data.t* < *threshold_t*
6. *temp_container.push_back*(*data*) // 过时数据存入临时容器
7. 标记原数据为无效 // 置 *y* 为无效, *t* 和 *time_last* 归零
8. else
9. *training_points.push_back*(*data*) // 有效数据加入训练集
   // 更新线性回归模型
10. *n* ← *training_points.size*()
11. if *n* > 0
12. 计算 $\sum x, \sum y, \sum xy, \sum x^2$
13. $denominator = n \times \sum x^2 - (\sum x)^2$
14. if $denominator \neq 0$
15. $leaf.k = (n \times \sum xy - \sum x \times \sum y)/denominator$

16. $leaf.b = (\sum y - leaf.k \times \sum x)/n$
17. $leaf.data\_num = n$
18. *merged* ← *merge_sorted*(*temp_container*, *cold_vector*) // 合并临时数据与冷数据
19. *cold_vector*.*swap*(*merged*) // 更新冷数据, 替换

数据插入过程的时间复杂度分析由两部分构成: 主路径定位插入与冷热数据迁移. 主路径中, 由根节点定位到叶子节点的总层级开销为 $\mathcal{O}(l \times \log error)$, 到达叶子节点后, 通过指数跳跃步长在预留间隙中探测插入位置, 耗时 $\mathcal{O}(\log(error))$, 且数据移动因间隙密度控制被约束为常数级 $\mathcal{O}(1)$, 整体主路径复杂度为 $\mathcal{O}((l+1)\times\log(error))$. 冷热数据迁移包含温度计算 $\mathcal{O}(N_{\text{seg}}\times T_{\text{calculate}})$、快速选择阈值 $\mathcal{O}(N_{\text{seg}})$、冷数据迁移 $\mathcal{O}(0.6\times N_{\text{seg}})$、热区增量训练 $\mathcal{O}(0.2\times N_{\text{seg}})$ 及冷区归并 $\mathcal{O}(0.6\times N_{\text{seg}} + M_{\text{seg}})$, 单次迁移总耗时 $\mathcal{O}(N_{\text{seg}} + M_{\text{seg}})$, 但通过高水位阈值 θ_{high} 异步触发与低频执行特性, 其开销分摊至单次插入可忽略.

2.3 数据查询过程

本研究采用冷热数据分区存储的创新架构, 突破了传统全数据拟合模式, 创新性地采用热区专属建模策略. 相较于传统全局建模方法, 本方案通过热区数据独立建模显著降低了模型预测误差, 并实现热区索引效率的提升. 针对冷区数据访问特征, 系统设计双层检索机制: 首先在热区建立高效索引查询, 未命中时触发冷区有序检索模块. 值得注意的是, 通过动态维护冷区数据的全局有序性, 冷区检索可基于有序结构的二分查找算法实现对数级时间复杂度检索. 实验结果表明, 该方案通过有限度降低冷区数据检索效率, 成功换取热区查询性能的阶跃式提升, 最终达成系统整体检索效率的优化.

对于给定查找键 k_{new}, 搜索其存储位置的过程如算法 3 所示. 首先要定位 k_{new} 所属的叶子节点, 从根节点加载初始线性回归模型 $M_j^{(1)}$ (第 8 行), 模型内部记录了其所辖键值空间的最小值 $\alpha_{\min}^{(j)}$ 与最大值 $\alpha_{\max}^{(j)}$ 边界. 计算目标键值 k_{new} 是否满足 $\alpha_{\min}^{(j)} \leqslant k_{\text{new}} \leqslant \alpha_{\max}^{(j)}$. 若满足包含条件, 则直接选定 $M_j^{(1)}$ 作为当前层级查询的终点; 反之则沿键值增大的方向 (若 $k_{\text{new}} > \alpha_{\max}^{(j)}$) 或减小的方向 (若 $k_{\text{new}} < \alpha_{\min}^{(j)}$) 进行指数查找, 直至定位到首个满足 $\alpha_{\min}^{(j+\Delta)} \leqslant k_{\text{new}} \leqslant \alpha_{\max}^{(j+\Delta)}$ 的新模型边界 $M_{j+\Delta}^{(1)}$ (第 10 行). 将选定的层级内模型 $M_{j+\Delta}^{(1)}$ 作为下一层查询的起点 (第 12 行), 递归执行上述范围判定与指数查找流程. 每经过一个层级 $l \to l-1$, 模型的粒度相应细化, 最终抵达叶子节点层 $l = 0$ (第 15 行).

算法 3. 搜索过程 *hierarchical_search*(*key*, *layers*).

输入: 搜索键 *key*, 层级结构 *layers*;
输出: 找到的数据指针或 null.

1. if *layers* is empty, return null
2. *current_model* ← null
3. for $l = L - 1$ down to 0 /* 从顶层 ($L - 1$) 向下遍历至叶子层 (0) */
4. if *current_model* is not null
5. *predicted* ← (*k* – *current_model.b*)/*current_model.k* /* 使用当前模型进行预测 */
6. Clamp *predicted* to [0, |*layer.models*| − 1] /* 将预测结果限制在合法范围 */
7. else
8. *predicted* ← (*k* – *root.b*)/*root.k* /* 使用根模型进行预测 */
9. Clamp *predicted* to [0, |*layer.models*| − 1]
10. *current_index* ← *exponentialModelSearch*(*layer.models*, *k*, *predicted*) /* 指数查找模型位置 */
11. if *current_index* == −1 return null
12. *current_model* ← *layer.models*[*current_index*] /* 选中当前层的模型 */
13. if *k* == *current_model.max_y*:

```
14.     current_model ← layer.models[current_index + 1] /* 特殊处理边界键值 */
15.   if l == 0 /* 若为叶子层 */
16.     predicted ← (k – current_model.b)/current_model.k /* 预测在叶子数据中的位置 */
17.     Clamp predicted to [0, |current_model.data_array| − 1] /* 限制预测位置范围 */
18.     pos ← exponentialSearch(current_model, k, predicted)
19.     if pos == −1
20.       pos ← binarySearch_cold(current_model.cold_zone, k) /* 冷区二分查找 */
21.       if pos != −1 return &current_model.data_array[pos] /* 找到返回指针 */
22.       else return null
23.     else
24.       if current_model.bitmap[pos] == false, print “false” /* 检查热数据有效性 */
25.       return &current_model.data_array[pos] /* 返回热区数据指针 */
26. return null
```

接下来对叶子节点进行指数搜索, 由叶子节点模型生成初始预测位置 p_{pred} (第 16 行), 当检测到该位置为空时, 则向相邻位置执行指数步长跳跃式探测直到找到首个非空位置, 通过对比目标键值 k_{new} 与该元素的数值关系, 选择正确的搜索方向执行指数搜索, 并基于间隙数组构建动态搜索窗口. 指数搜索得到正确的搜索窗口后, 采用边界迭代收缩策略, 通过不断更新窗口的左右边界索引, 最终确定目标数据的实际位置 (第 18 行). 若通过上述过程未找到数据 k_{new}, 则最后对冷区进行二分查找 (第 20 行). HCA-Index 从节点粒度构建冷区, 所以可以通过叶子节点与冷区的一一映射关系直接确定相应冷区 C_{cold}, 在冷区中执行二分查找, 如果搜索仍未命中, 则返回查找失败 (第 21、22 行). 对于冷区中的数据, 其上次被访问的时间戳会被记录. 若相邻两次访问的时间间隔短于预设值, 则计为一次连续访问, 其次数会相应累加. 一旦连续访问次数超过既定阈值, 则触发数据迁移逻辑, 将该数据从冷区提升至热区.

基于 HCA-Index 的数据查询过程时间复杂度分析如下, 查找的时间开销来自两部分: 一部分是基于热区的查找, 另一部分是冷区的二分查找. 查找数据时, 层级模型树通过预定义层数 l 逐层缩小键值范围, 每层基于预测模型与指数查找定位子节点的时间复杂度为 $\mathcal{O}(\log(error))$, 其中 $error$ 为预测位置与实际边界的偏差, 故总层级开销为 $\mathcal{O}(\log(error)\times l)$; 到达叶子节点后, 通过基于空隙数组的指数查找定位数据位置, 耗时为 $\mathcal{O}(\log(error))$. 如果未能在热区搜索到待查找数据, 则花费 $\mathcal{O}(\log(N_{\text{seg}}))$ 时间在冷区执行二分查找. 插入新数据时仅仅会影响节点中 N_{seg} 的大小, 而不会改变索引整体的结构, 甚至随着热区中的数据迁移再训练, 预测误差 $error$ 能够进一步降低, 因此, 插入操作对热区数据的查询速度几乎没有影响. 随着 N_{seg} 的增大, 冷区的搜索时间可能增加, 但是二分搜索的时间为对数级别, 因此 N_{seg} 的增大并不会显著增加冷区搜索时间. 总体来看, 由于热区数据的查找概率要远高于冷区数据, 数据查找的平均时间主要取决于热区数据的查找时间, 插入对本模型的搜索效率影响甚微.

3 实验分析

3.1 实验设置与数据集

本节设计全面的实验验证 HCA-Index 索引在查询响应时间、索引内存开销以及索引构建时间的效果. 实验在一台采用第 13 代 Intel Core i9-13900HX 处理器 (5.4 GHz 睿频, 24 核 32 线程) 与 NVIDIA GeForce RTX 4060 显卡的计算机上进行, 基于 C++17 标准实现了 HCA-Index 以及所有的对比算法. 所有对比实验均在禁用超线程与 Turbo Boost 技术的控制条件下执行, 程序的优化等级为-O2. 实验在 SOSD 公开的 4 个标准数据集 (osm、wiki、fb、books) 上进行, 这些数据集的具体情况如表 2 所示, 其中, osm 基于地理坐标 (均匀分布含局部聚类), wiki 来

自维基百科时间戳 (递增时序数据), fb 为合成社交网络数据 (高度偏态分布), books 取自亚马逊商品信息 (稀疏与密集混合分布), 每个数据集包含 2–8 亿个整数, 旨在系统评估学习索引在不同数据分布下的查询性能及鲁棒性, 覆盖均匀、时序、偏态和复杂混合场景的测试需求. 为了统一实验标准, 在后续实验中均截取数据集中 200M 个数据进行实验 (wiki 数据集中存在重复键值对, 部分索引不支持重复键值对插入, 对此数据集进行重复值去除处理后截取其中 80M 数据进行实验). 实验采用 8 字节键值与固定长度的随机生成有效载荷构成键-值对, 进行索引性能测试.

表 2　实验数据集

数据集	数据数量	键类型	有效载荷 (B)	数据集大小 (GB)
osm	1B	double	8	16
wiki	200M	uint64	8	1.6
fb	200M	double	8	2.4
books	800M	uint64	8	1.7

3.2　工作负载与基准算法

为了验证新数据插入对 HCA-Index 对数据插入的影响, 设计 5 组差异化读写配比的工作负载 (读写配比分别为 0、0.2、0.4、0.6、0.8). 其中, 纯读模式 (读写配比 0) 通过 20M 次连续读操作模拟读密集场景下的极限性能, 后续 4 组混合负载 (读占比 20%-80%) 各执行 20M 次混合操作, 通过不同配比分析系统在动态负载下的性能衰减规律. 同时, 为了比较 HCA-Index 相比于传统索引 (即 B+ tree) 和 6 种代表性的学习型索引在查询响应时间、索引内存开销以及索引构建时间的性能, 分别选取 B+ tree、RMI、RadixSpline、ALEX、PGM-index、LIPP、SWIX 和 RadixSpline 作为基准算法, 这些基准算法的特点见表 1. 其中 RMI 和 RadixSpline 不支持插入操作, 因此本实验仅在只读场景下对其进行测试.

3.3　实验结果与分析

本节通过 3 项核心指标对算法性能进行横向对比分析. 查询响应时间: 基于多样化数据分布 (均匀/时序/偏态) 测试不同情况下索引的点查询平均延迟, 此项指标反映了在密集查询情况下索引的查询性能. 索引内存开销: 测量各类索引结构在内存中的物理占用规模. 索引构建时间: 对比不同索引在相同数据集上的构建速度以及相同索引对不同数据集构建的适应能力. 查询吞吐量: 吞吐量指系统在单位时间内的操作处理量, 是衡量不同读写比例下索引综合性能的关键指标. 同时, 在查询响应时间的测试实验中, 还设计了 5 组差异化读写配比的工作负载 (读写配比分别为 0、0.2、0.4、0.6、0.8), 用于测试新数据涌入对 HCA-Index 查询性能的影响.

3.3.1　查询响应时间

图 6 展示了 HCA-Index 与 7 种基准索引在 4 种不同数据分布的数据集上进行 2 000 万次查询的总耗时对比. HCA-Index 在所有数据集上均展现出显著的性能优势, 其查询效率分别超越 B+ tree、RMI、RadixSpline、ALEX、PGM-index 和 LIPP 等基准索引. 这种性能优势主要源于两个方面: 首先, 基于误差控制的渐进式分区算法通过动态适应数据分布特征, 有效降低了模型拟合误差; 其次, 键值范围层级合并策略通过减小上层模型的拟合误差以及扩大上层模型命中范围, 进一步优化了整体索引结构的查询响应时间. 值得注意的是, 在不同分布特性的数据集上, HCA-Index 的查询耗时波动范围显著小于其他对比索引, 这验证了该索引架构对数据分布差异具有较强的鲁棒性, 这得益于键值范围层级合并的策略缩小了原始数据分布对上层模型构建的影响, 以及基于误差阈值的渐进分区算法对分区误差的保障. 具体而言: 在局部聚类数据集 osm 上, HCA-Index 的查询时间分别比 B+ tree、RMI、RadixSpline、ALEX、PGM-index、LIPP、SWIX 降低了 90.42%、62.30%、80.83%、77.00%、74.44%、69.33%、84.56%. 在高度偏态分布数据集 fb 上, 由于数据分布的特点, HCA-Index 的查询时长有所增加, 但仍然比 B+ tree、RMI、RadixSpline、ALEX、PGM-index、LIPP、SWIX 降低了 86.61%、31.91%、81.92%、68.00%、62.35%、67.01%、57.89%. 这些实验数据充分证明了本文方法在复杂数据场景下的优越适应能力.

HCA-Index 在不同插入比例下的平均查找时间变化趋势如图 7 所示, 在 4 个不同数据分布的数据集上, 随着

插入比例的增大, 系统的平均查找时间呈现总体下降趋势. 这是由于本文提出的动态温度冷热识别模型和冷热数据迁移策略发挥了作用: 低概率访问数据被迁移至冷区, 而 HCA-Index 通过专注拟合热区的高概率访问数据, 有效降低了热区数据的拟合误差, 从而提升了热区数据的查询效率. 虽然冷区数据的查询速度有所下降, 但是由于热区数据更高频次被访问, 系统整体查询时间仍保持下降态势. 值得关注的是, 在某些特定插入比例区间内, 平均查找时间出现短暂上升. 这是因为新插入数据尚未达到冷热迁移阈值, 导致冷热数据暂时共存于热区, 而间隙插入方式会改变热点数据的分布位置, 造成查询误差增大, 进而对查询速度产生暂时性影响.

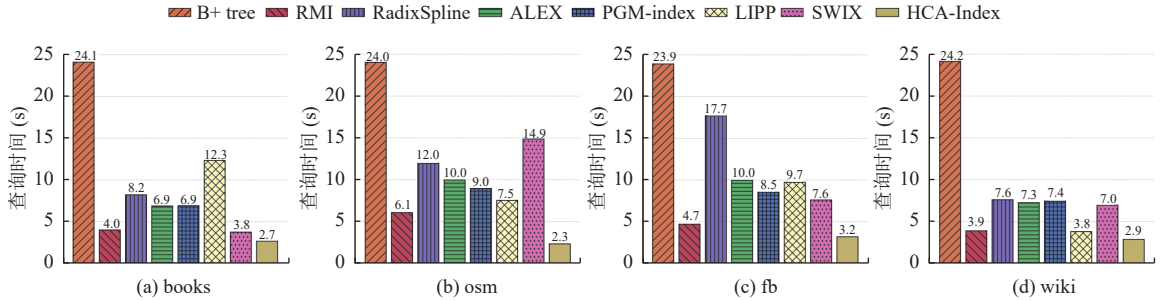


图 6　查找时间对比

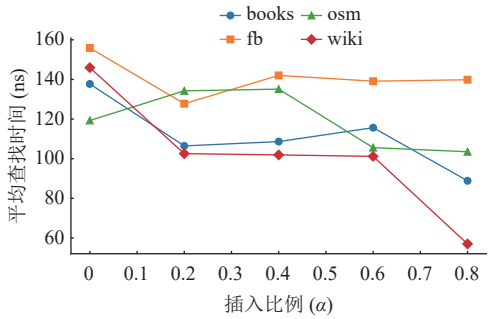


图 7　HCA-Index 查找延迟随插入比例变化趋势

3.3.2　索引内存开销

HCA-Index 与 7 种基准索引在 4 种不同数据分布数据集上的索引内存开销如图 8 所示, HCA-Index 在所有数据集上均展现出显著的空间效率优势, 其索引规模较 B+ tree、RMI、RadixSpline、ALEX 和 PGM-index 等基准索引分别实现大规模缩减. 具体而言, 在局部聚类数据集 osm 中, HCA-Index 的索引规模较对比索引分别缩减 99.53% (B+ tree)、95.76% (RMI)、97.94% (RadixSpline)、97.21% (ALEX)、95.40% (PGM-index)、99.73% (LIPP)、96.55% (SWIX). 该数据集呈现的局部聚类特性使底层分区范围呈现高密度分布特征, 有利于渐进式分区算法实现更大范围的数据覆盖. 值得关注的是, 在高度偏态数据集 fb 中, 虽然极端偏态分布导致数据点离散度增加, 使得渐进式分区算法需要创建更多底层分区 (相较 osm 数据集构建得到的索引内存占用增加了 4.18×), 但 HCA-Index 的索引规模仍较基准索引实现显著降低, 具体缩减倍数分别为 98.02% (B+ tree)、82.26% (RMI)、91.70% (RadixSpline)、88.32% (ALEX)、83.86% (PGM-index)、98.88% (LIPP)、78.78% (SWIX). HCA-Index 内存开销降低可以归因于两方面: (1) 基于误差控制的渐进式分区算法通过动态适应数据分布特征, 显著提升了分区覆盖范围, 在保证误差阈值的前提下将分区数量减少, 从而降低模型总量; (2) 键值范围层级合并策略通过自底向上的分区范围聚合, 有效压缩了上层索引结构的空间复杂度. 综上, HCA-Index 相比于基准算法在索引内存开销方面具有显著优势 (本实验构建时均取 0.8 倍数据集总量, 其中 B+ tree 与 LIPP 索引在高数据量时内存占用过大, 均固定取 80M 数据量进行训练).

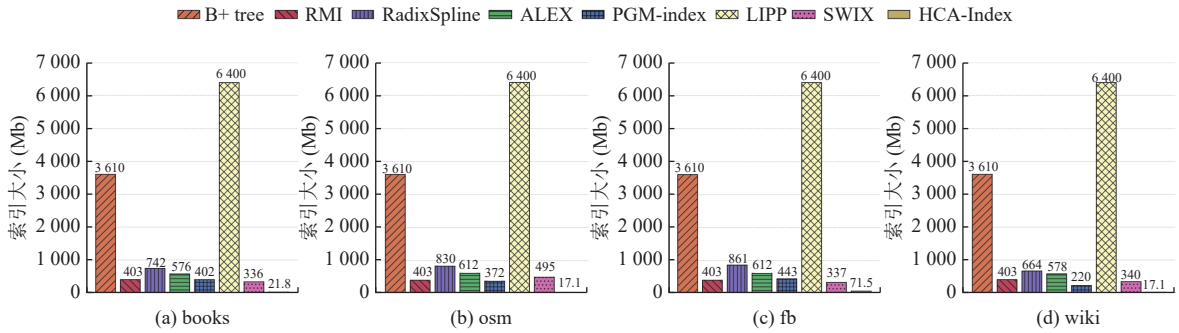


图 8 索引大小对比

3.3.3 索引构建时间

图 9 展示了 HCA-Index 与 7 个基准测试索引的构建时间对比. 尽管 B+树的索引构建时间极短, 但是如图 6 和图 8 所示, 其查询响应时间相比于学习型索引相差很大, 其索引内存开销也高于绝大多数学习型索引. 在学习型索引当中, HCA-Index 在所有数据集上的表现均优于 RMI, 同时, HCA-Index 在多个数据集上的索引构建时间相比于 PGM-index 和 ALEX 更优或相近. 尽管 HCA-Index 比 RadixSpline 和 SWIX 的索引构建时间更长, 但是综合考虑 HCA-Index 在查询响应时间和索引内存开销的优势, 索引构建时间更长在一定程度上是足以容忍的.

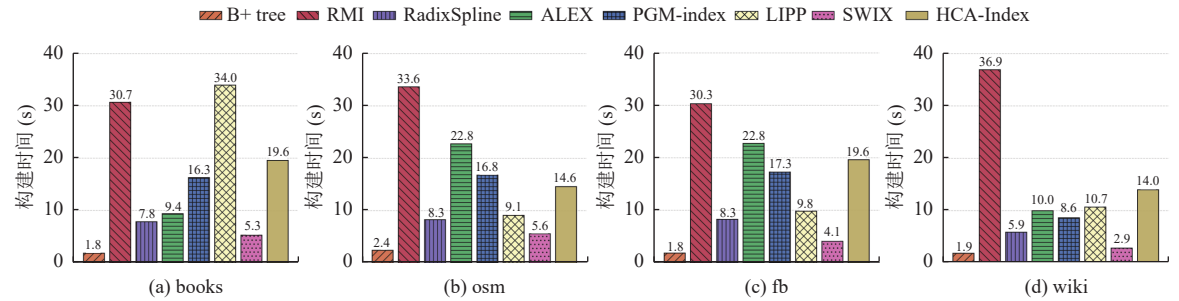


图 9 构建时间对比

为了进一步探究 HCA-Index 索引构建时间较大的原因, 进一步设计实验测量两个索引构建过程的时间: (1) 叶子层构建时间, 用于反映最底层运行渐进分区算法的时间占用. (2) 总构建时间. 结合这两个时间可以计算出层级合并策略构建上层模型的时间占用情况. 表 3 展示了在 4 个不同数据集上, 分别使用 200M、100M 和 20M 数据量构建索引所需的总时间及其叶子层构建时间. 分析表 3 可知, 在所有情况下, 叶子层构建时间均占据了总构建时间的 99% 以上. 这表明本文提出的、基于范围层级合并的上层索引构建方法确实有效, 成功将上层索引的构建耗时减少了数个数量级. 因此, 性能瓶颈在于构建底层分区模型的渐进分区算法 (progressive partitioning algorithm, PPA) 耗时过高. 尽管通过增量学习优化已将该算法的理论复杂度降至 O(n) 级别, 但其实际构建时间较长.

进一步观察 wiki 时序数据集上 200M 与 100M 数据量对应的总构建时间, 结果显示: 在数据量仅增加 2 倍的情况下, 构建时间却增长了 5.24×. 在其他数据集上, 该增幅也分别达到了 4.1×、3.16×和 6.37×. 首先推测, 数据分布差异可能导致渐进分区算法运行速度变化, 进而引发这种非线性的时间增长. 然而, 后续实验将 200M 数据中的后 100M 单独构建索引, 并将其耗时与前 100M 的构建时间相加, 结果仍显著小于直接在 200M 数据上完整构建的耗时. 由此可见, 数据分布变化并非导致运行速度显著下降的主导因素. 更可能的原因在于处理高数据量时, 涉及的 CPU 缓存失效、内存频繁分配与释放等系统层面的开销.

针对以上问题, 提出批量加载 (batch loading) 策略优化索引的构建时间, 即分批次将数据加载入内存进行构建, 而非一次性加载全部数据. 以 wiki 数据集为例: 采用批量加载策略分批加载 10 次 (共 200M 数据) 的总耗时为 5 607 452 701 ns. 相比直接加载全部 200M 数据的构建时间 14 664 395 240 ns, 此方法可将耗时缩减超过 60%, 显著降低了总构建时间. 同时, 构建后的索引在查询响应时间和索引内存开销方面仍能保持上文提到的优势. 如图 10

所示, 优化后的 HCA-Index 在所有数据集上的构建时间均显著优于 RMI 和 PGM-index; 在多数数据集上也优于 ALEX 和 LIPP, 但略逊于 SWIX, 与 RadixSpline 则基本相当.

表 3　构建时间分析

数据集	数据量 (M)	总构建时间 (ns)	叶子层时间 (ns)	占比 (%)
wiki	200	14 664 395 240	14 647 994 773	99.89
	100	2 801 067 900	2 798 255 167	99.90
	20	962 771 733	962 247 866	99.95
books	200	18 070 956 900	18 051 242 200	99.89
	100	4 409 867 833	4 408 273 800	99.96
	20	453 540 600	453 469 933	99.98
osm	200	14 599 349 233	14 587 699 100	99.92
	100	4 616 124 067	4 611 217 500	99.89
	20	979 188 633	978 684 266	99.95
fb	200	20 563 717 100	20 512 300 733	99.75
	100	3 229 700 233	3 215 104 400	99.55
	20	662 268 866	659 305 966	99.55

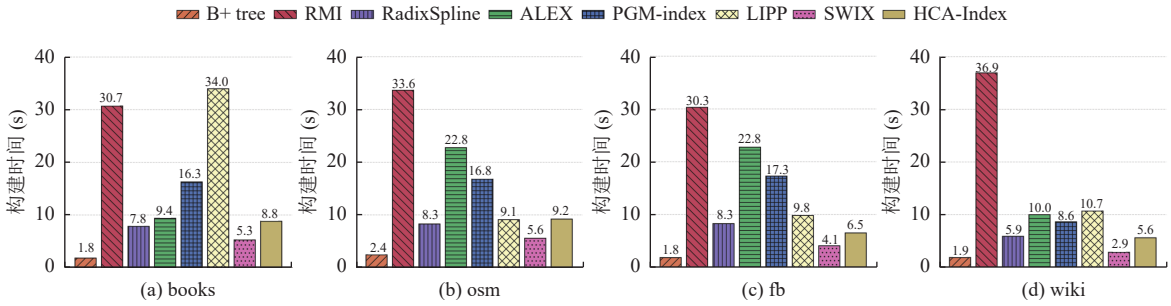


图 10　优化后构建时间对比

3.3.4　索引吞吐量

本实验通过 Zipf 分布分批次生成具有时变特性的查询热点, 以模拟真实场景中热点的动态变化, 并在此基础上对比了不同索引结构在包含显著热点数据的多种读写负载下的吞吐性能. 如图 11 所示, 在写入比例较低时, HCA-Index 由于能够准确识别热点数据, 在持续写入大量新数据的同时仍保持热点数据的高效查询, 因而呈现出显著的吞吐量优势. 然而, 随着写入比例的增加, 频繁的写入操作会触发数据迁移过程, 而在迁移期间, 后续的查询与写入请求会被阻塞, 导致系统吞吐量明显下降. 通过合理设置迁移操作的触发阈值, 可有效降低迁移频率, 从而改善整体吞吐性能. 在写入比例较高的场景中, HCA-Index 因预设了充足的存储空间间隙, 并具备高效的插入位置定位机制, 其吞吐量仍保持一定优势. 仅在纯写入 (write-only) 负载下, 其吞吐量略低于 PGM-index.

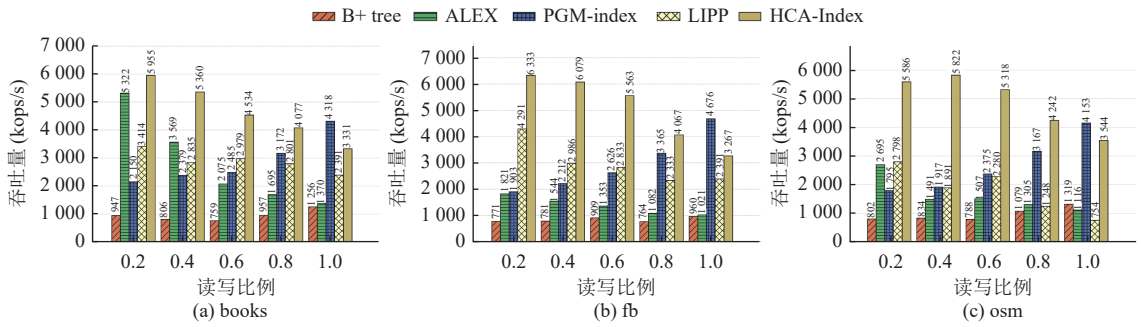
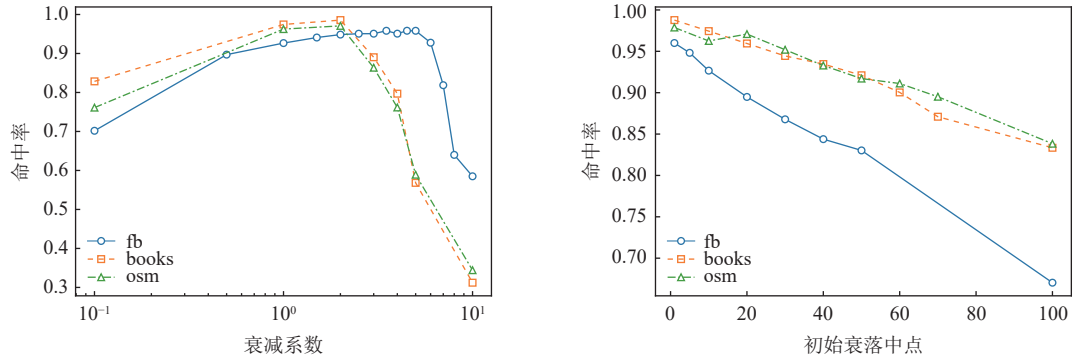


图 11　不同读写比的索引吞吐量对比

3.4 参数设置测试

在动态温度识别模型 (dynamic thermal recognition model, DTRM) 中, 多个关键参数的设置会显著影响热区命中率, 进而对整体索引性能产生重要影响; 同时, 热区数组中数据迁移阈值的设定也会显著影响数据迁移频率与插入操作效率. 为此, 本节通过对这些参数开展灵敏度测试, 评估其具体影响, 以确定适宜的参数取值范围. 在考察温度参数对热区命中率的影响时, 由于参数数量较多且单次实验耗时较长, 本文通过多次实验筛选出一组能够取得较高热区命中率的参数配置作为基准设置, 具体为: 衰减系数为 1, 基础加热值为 50, 加热系数为 50, 初始衰减中点为 10. 在后续分析单一参数对命中率的影响时, 均基于该默认参数组合进行.

图 12 展示了在默认参数设置下, 衰减系数与初始衰减中点对热区命中率的影响. 从图中可以看出, 在 3 个数据集中, 当衰减系数接近 10 时, 命中率均呈现明显下降趋势. 这是因为 DTRM 在计算数据热度时, 温度的衰减模式通常为先慢后快再转缓, 过大的衰减系数将导致温度下降过快, 使得热点数据在较短时间内降至与冷数据相近的温度水平, 从而削弱冷热数据的区分能力. 相反, 若衰减系数过小, 热点数据温度下降缓慢, 新生成的热点数据难以在温度上超越旧热点, 导致模型无法有效识别数据访问的动态变化, 命中率也因此下降. 综合 3 个数据集的实验结果, 当衰减系数处于 1–3 的范围内时, 热区命中率均能维持在 0.9 以上, 且波动较小. 因此, 将衰减系数设定在 1–3 之间较为合理.



(a) 衰减系数对命中率的影响 (b) 衰减中点对命中率的影响

图 12 衰减系数与衰减中点对命中率影响

初始衰减中点决定了温度开始快速衰减的时机. 图示结果表明, 当该参数接近 100 时, 命中率出现明显下降. 这是因为过大的初始衰减中点会延缓温度衰减过程, 导致旧热点数据的温度持续高于新热点, 影响模型对动态访问模式的响应能力, 从而降低命中率. 当初值衰减中点处于 0–20 的区间内时, 热区命中率均保持在 0.9 以上. 由此可见, 在衰减系数设置合理的情况下, 初始衰减中点可在较宽的范围内选取.

表 4 和表 5 分别展示了在 3 个数据集中, 不同加热值与加热系数的设定对热区命中率的影响. 从结果可以看出, 在 1–100 的范围内, 加热值和加热系数对命中率的影响均不显著, 命中率始终维持在 0.9 以上. 然而, 当加热值为 0 时, 命中率出现明显下降. 这是由于在缺乏加热项的情况下, 数据即使被频繁访问, 其温度也无法有效提升, 导致模型难以区分冷数据与热数据, 从而影响识别效果.

图 13 展示了在 3 个数据集中, 迁移阈值 (即高水位阈值) 从 0.65 变化至 0.95 时, 索引吞吐量的变化情况. 实验结果表明, 阈值设定过高或过低均会导致吞吐量显著下降. 当阈值接近 0.65 时, books、fb 和 osm 这 3 个数据集中的吞吐量分别下降至峰值水平的 54%、61% 和 68%. 阈值设置过低会大幅增加数据迁移频率, 从而阻塞查询与插入过程, 降低整体吞吐性能. 而当阈值接近 0.95 时, 3 个数据集中的吞吐量分别为峰值水平的 72%、71% 和 75%. 过高的阈值虽然减少了迁移次数, 但会导致热区剩余空间不足, 引发插入时的频繁数据移动, 同时冷数据无法及时迁出也会降低热区数据的查询效率. 综合实验数据, 将阈值设置在 0.75–0.85 之间时, 索引吞吐量整体表现较优.

表 4 加热值对命中率影响

加热值	fb	books	osm
0	0.491 124	0.024 111	0
25	0.967 838	0.976 633	0.960 48
50	0.959 984	0.974 563	0.959 001
75	0.960 513	0.976 505	0.959 15
100	0.937 885	0.972 119	0.970 978

表 5 加热系数对命中率影响

加热系数	fb	books	osm
1	0.938 134	0.980 791	0.971 384
25	0.962 952	0.980 933	0.959 937
50	0.961 803	0.974 147	0.962 814
75	0.961 364	0.976 782	0.960 686
100	0.926 103	0.974 818	0.970 841

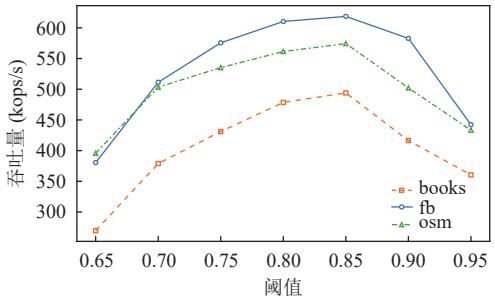


图 13 高水位阈值对吞吐量影响

3.5 消融实验

为验证动态温度识别模型 (DTRM) 在存在显著热点数据场景下对索引性能的提升效果, 本节设计了消融实验以评估其作用. 实验中, 迁移策略仍沿用原索引机制, 但在迁移过程中不再依据数据热度, 而是采用随机迁移方式. 实验结果如图 14 所示. 在 3 个数据集中, 当写入比例较低时, 消融后的索引吞吐量较原始方法出现明显下降. 这是由于模型失去冷热识别功能后, 大量实际热点数据被随机迁移至冷区, 而冷区的查询效率低于热区, 从而拉低了整体查询性能. 在纯写入 (write-only) 负载下, 由于不涉及查询操作, 消融前后的吞吐量基本保持一致.

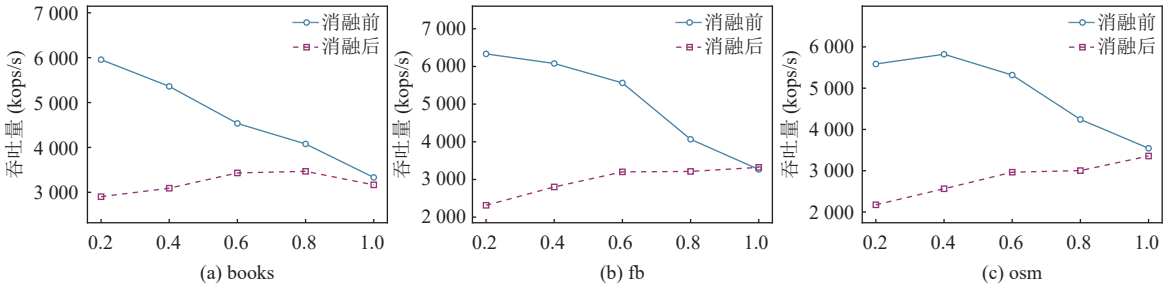


图 14 消融实验结果

4 结 论

本文针对学习型索引难以适应不同数据分布, 以及大量新数据插入导致的索引性能退化问题, 提出了一种基于冷热数据感知的学习型索引 HCA-Index. 该索引通过基于误差阈值的渐近分区算法, 有效适应不同数据分布; 采用自下而上提取键值范围的层级合并策略构建索引, 在提升精度的同时实现了索引的轻量化与快速构建; 通过动态演化的数据温度计算模型识别冷热数据, 相较于先前学习型索引, 充分利用了数据的冷热属性, 从而提高了索引性能; 设计节点级冷热分区实现了数据的快速迁移, 同时加速了热区重建. 在多个真实数据集上的对比实验表明, 相较于现有的学习型索引, HCA-Index 在显著降低索引大小的同时, 大幅提升了索引构建与查询速度, 提升了吞吐量, 并有效缓解了数据插入导致的索引性能退化.

References

[1] Bingmann T. STX B+ tree C++ template classes. 2013. https://panthema.net/2007/stx-btree/

[2] Stanford DAWN Cuckoo Hashing. 2017. https://github.com/stanford-futuredata/index-baselines

[3] Guo N, Wang YQ, Jiang HN, Gu Y, Xia XF. DRAMA: Update-distribution-aware learned index. Ruan Jian Xue Bao/Journal of Software, 2025, 36(8): 3769–3786 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7220.htm [doi: 10.13328/j.cnki.jos.007220]

[4] Zhang SM, Cai P, Li CP, Chen H. High-dimensional learned index based on space division and dimension reduction. Ruan Jian Xue Bao/Journal of Software, 2023, 34(5): 2413–2426 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6414.htm [doi: 10.13328/j.cnki.jos.006414]

[5] 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]

[6] Tang CZ, Dong ZY, Wang MJ, Wang ZG, Chen HB. Learned indexes for dynamic workloads. arXiv:1902.00655, 2019.

[7] Zhang JY, Gao YH. CARMI: A cache-aware learned index with a cost-based construction algorithm. Proc. of the VLDB Endowment, 2022, 15(11): 2679–2691. [doi: 10.14778/3551793.3551823]

[8] Yu T, Liu GF, Liu A, Li ZX, Zhao L. LIFOSS: A learned index scheme for streaming scenarios. World Wide Web, 2023, 26(1): 501–518. [doi: 10.1007/s11280-022-01021-6]

[9] Yang G, Liang L, Hadian A, Heinis T. FLIRT: A fast learned index for rolling time frames. In: Proc. of the 26th Int'l Conf. on Extending Database Technology. Ioannina: OpenProceedings.org, 2023. 234–246. [doi: 10.48786/edbt.2023.19]

[10] 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]

[11] 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]

[12] 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]

[13] Crotty A. Hist-tree: Those who ignore it are doomed to learn. 2021. https://cs.brown.edu/people/acrotty/pubs/cidr2021_paper20.pdf

[14] Leis V, Kemper A, Neumann T. The adaptive radix tree: ARTful indexing for main-memory databases. In: Proc. of the 29th IEEE Int'l Conf. on Data Engineering (ICDE). Brisbane: IEEE, 2013. 38–49. [doi: 10.1109/ICDE.2013.6544812]

[15] Kipf A, Marcus R, van Renen A, Stoian M, Kemper A, Kraska T, Neumann T. RadixSpline: A single-pass learned index. In: Proc. of the 3rd Int'l Workshop on Exploiting Artificial Intelligence Techniques for Data Management. Portland: ACM, 2020. 5. [doi: 10.1145/3401071.3401659]

[16] Setiawan NF, Rubinstein BIP, Borovica-Gajic R. Function interpolation for learned index structures. In: Proc. of the 31st Australasian Database Conf. on Databases Theory and Applications. Melbourne: Springer, 2020. 68–80. [doi: 10.1007/978-3-030-39469-1_6]

[17] 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]

[18] 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]

[19] Wang QG, Zhang YF, Wang H, Geng L, Lee R, Zhang XD, Yu G. Automating incremental and asynchronous evaluation for recursive aggregate data processing. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 2439–2454. [doi: 10.1145/3318464.3389712]

[20] Li PF, Lu H, Zhu R, Ding BL, Yang L, Pan G. DILI: A distribution-driven learned index. Proc. of the VLDB Endowment, 2023, 16(9): 2212–2224. [doi: 10.14778/3598581.3598593]

[21] Gao YN, Ye JB, Yang NZ, Gao XF, Chen GH. Middle layer based scalable learned index scheme. Ruan Jian Xue Bao/Journal of Software, 2020, 31(3): 620–633 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5910.htm [doi: 10.13328/j.cnki.jos.005910]

[22] Liang L, Yang G, Hadian A, Croquevielle LA, Heinis T. SWIX: A memory-efficient sliding window learned index. Proc. of the ACM on Management of Data, 2024, 2(1): 41. [doi: 10.1145/3639296]

附中文参考文献

[3] 郭娜, 王雅琪, 姜皓南, 谷峪, 夏秀峰. DRAMA: 更新分布感知的学习型索引. 软件学报, 2025, 36(8): 3769–3786. http://www.jos.org.

cn/1000-9825/7220.htm [doi: 10.13328/j.cnki.jos.007220]
[4] 张少敏, 蔡盼, 李翠平, 陈红. 基于区域划分与降维的高维学习型索引. 软件学报, 2023, 34(5): 2413–2426. http://www.jos.org.cn/1000-9825/6414.htm [doi: 10.13328/j.cnki.jos.006414]
[21] 高远宁, 叶金标, 杨念祖, 高晓泓, 陈贵海. 基于中间层的可扩展学习索引技术. 软件学报, 2020, 31(3): 620–633. http://www.jos.org.cn/1000-9825/5910.htm [doi: 10.13328/j.cnki.jos.005910]

作者简介

孙伊鸣, 本科生, 主要研究领域为数据库系统, 学习型索引, 事务处理机制.
信俊昌, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为大数据管理与分析, 多模态数据分析, 医学信息学.
赵少杰, 本科生, 主要研究领域为学习型索引, 数据库系统, 一致性机制.
操天童, 本科生, 数据库系统, 学习型索引, 事务处理机制.
洪振强, 本科生, CCF 学生会员, 主要研究领域为学习型索引.
安宸, 本科生, 主要研究领域为大数据管理与分析.